UCRL-MA-153515

Parallel Dislocation Dynamics DD3d User's Manual

M. Bartelt, V. Bulatov, W. Cai, M. Hiratani,
T. Pierce, M. Rhee, M. Tang

June 4, 2003

Parallel Dislocation Dynamics DD3d User's Manual

Lawrence Livermore National Laboratory
7000 East Avenue, Livermore, CA 94550,USA

Contributors:

(*in alphabetical order*)

Maria Bartelt (CMS), bartelt3@llnl.gov, (925) 422-7259

Vasily Bulatov (CMS), bulatov1@llnl.gov, (925) 423-0124

Wei Cai (CMS), caiwei@llnl.gov, (925) 424-5443

Masato Hiratani (CMS), hiratani1@llnl.gov, (925) 423-0896

Tim Pierce (CASC), pierce7@llnl.gov, (925) 423-6900

Moono Rhee (NTED), rhee1@llnl.gov, (925) 424-4990

Meijie Tang (PAT),tang7@llnl.gov, (925) 422-2415

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

June 4, 2003

Contents

1 Data Structure

In DD3d dislocations are represented by nodes that are interconnected by straight segments, as shown in Fig. 1. The nodes can be discretization nodes for representing a smooth line (e.g. node 1, 2, 3) or a physical node where three dislocations meet (e.g. node 0). Both discretization and physical nodes are treated on equal footing in DD3d — they have a common data structure and essentially the same equations of motion. The position of nodes can be any real number, i.e. they do not sit on a lattice. Two nodes connected by an arm are called neighbors. A node can have arbitrary number of neighbor nodes. The arms between nodes represent straight dislocation segments, hence are associated with a Burgers vector. The Burgers vector on each arm is fixed until it is destroyed or altered due to a dislocation reaction.

The information for each arm is redundantly stored on the two end nodes. The Burgers vector is defined along the line direction that points from the current node (that stores the Burgers vector) to its neighboring node. Therefore, the Burgers vectors for every arm stored on its two end nodes are the opposite of each other, e.g. $\vec{b}_{01} + \vec{b}_{10} \equiv 0$. With this convention, the sum of all Burgers vectors stored in every node must be zero as well, e.g. $\vec{b}_{01} + \vec{b}_{02} + \vec{b}_{03} \equiv 0$.

The DD3d simulation proceeds by integrating the equations of motion for all the nodes, i.e. updating the nodal positions at every time step. New nodes may be created and old nodes may be deleted during the simulation, to reflect the physical creation and annihilation of dislocation lines, and to maintain a faithful yet efficient discretization of the evolving dislocation network.

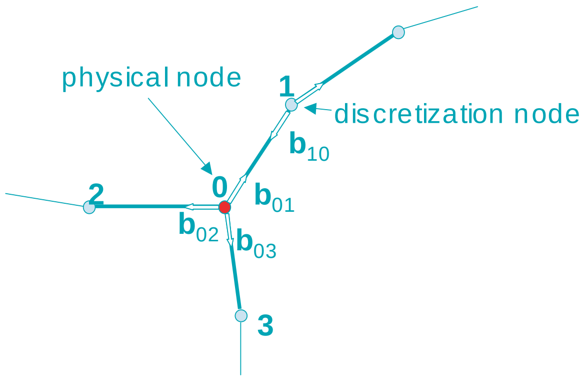


Figure 1: Nodal representation of dislocation network in DD3d (see text). $\vec{b}_{01}$ is the Burgers vector for the directed arm from node 0 to node 1.

1.1 Nodes

In `Node.h` the data structure for nodes are defined as follows.

```
typedef struct _node Node_t ;
struct _node {
   ...
   real8 x, y, z ;   /* nodal position */
   real8 fX, fY, fZ ;/* nodal force */
   real8 vX, vY, vZ ;/* nodal velocity */

   real8 oldx, oldy, oldz ;  /* node position in last step */
   ...
   Tag_t myTag ;    /* Tag (domainID,index) for node */
   int numNbrs ;    /* number of neighbor nodes */
   Tag_t *nbrTag ; /* pointers of neighbor node tags */
```

```
    real8 *burgX, *burgY, *burgZ;/* burgers vector */
    real8 *nx, *ny, *nz;         /* glide plane */
    real8 *ux, *uy, *uz, *ur;    /* line direction and length */
    ...

    int constraint ; /* type of constraint on node */

    int cellIdx ;   /* cell node is currently sorted into */
    int cell2Idx;
    int native ;    /* 1 = native node, 0 = ghost node */

    Node_t *next ;  /* pointer to next node in queue (ghost or free) */
    Node_t *nextInCell ; /* used to queue node onto current containing cell */

    Node_t *nextInCell2 ; /* used to queue node onto (secondary) cell for
                             collision detection */
} ;
```

For each node, $\vec{r}$ =(`x, y, z`) specifies its position; $\vec{f}$ =(`fX, fY, fZ`) is the force acting on it; and $\vec{v}$ =(`vX, vY, vZ`) is its velocity in response to this force. (`oldx, oldy, oldz`) stores the nodal position in the last time step. They are used to determine how much a node has moved, for computing the incremental plastic strain in the current time step.

The `myTag` entry of specifies the identification of the node. A `Tag_t` is a structure of two integers – (`domainID, index`), where `domainID` specifies which domain (each domain is handled by a separate CPU, see next section) the node belongs to, and `index` specifies the node's index within that domain. Each node can have arbitrary number of neighbors. The total number of neighbors is given in `numNbrs` (a positive integer), and the tags of neighboring nodes are stored in array `*nbrTag`.

The Burgers vectors of each arm $\vec{b}_j$ ($j = 0, \cdots,$ `numNbrs` -1) are stored in three arrays (`*burgX, *burgY, *burgZ`). Similarly, the glide plane normal $\vec{n}_j$ of each arm are stored in (`*nx, *ny, *nz`). The unit vectors for each arm's line direction are stored in (`*ux, *uy, *uz`), and length of each arm are stored in `*ur`. The integer `constraint` specifies the type of constraint on the node: e.g. 0 means no constraint and 7 means the node is fixed.

For computational efficiency the simulation space is also divided into cells (see next section). `cellIdx` specifies which cell the node belongs to. Nodes belong to the same cell also forms a queue for fast referencing. The pointer `*nextInCell` points to the next node in each queue. `cell2Idx` and `*nextInCell2` are for the second (independent) cell structure and have similar meaning as `cellIdx` and `*nextInCell`.

Nodes that are located within the physical boundary of a domain are called *native* nodes to this domain. Others are called *ghost* nodes to this domain. When DD3d runs in parallel, each CPU (corresponding to one domain) stores the information of all of its native nodes, plus the information of some ghost nodes, which are native to other domains surrounding this domain. `native`= 1 if the node is native; `native`= 0 if it is ghost. All native nodes form a queue; all ghost nodes form another queue. The pointer `*next` points to the next (native/ghost) node in these queues.

Therefore, there are multiple ways to enumerate the nodes. One way is to loop through all the tags. One can also traverse the native node queue and ghost node queue. One can also loop through all the cells (or `cell2`s) and traverse the node queue on each cell (or `cell2`).

1.2 Domains

DD3d is a parallel code based on MPI (Message Passing Interface). Except in the initialization phase, all processors are equivalent, i.e. there is no distinctions such as "master" and "slave" between processors. The entire simulation space is a 3-dimensional rectangular box, which is divided into different "domains". This decomposition is done sequentially along x, y, z axises (in `Decomp.c`). When a text input file is used, CPU 0 reads in the entire nodal structure and decompose the data into domains. First the entire space is divided into `numXdoms` chunks along the x direction such that each chunk contains equal number of nodes. Each chunk is then divided along y direction by `numYdoms` times, with the resulting chunk again divided along z direction by `numZdoms`. Each chunk after this decomposition is a domain, as illustrated in Fig. 2(a). CPU 0 then inform other CPUs, each in charge of one domain, of the information on their native nodes. After that, each CPU only exchange nodal information with its neighboring CPUs (domains) during the simulation. Ideally, the number of nodes per domain should be around 200-1000.

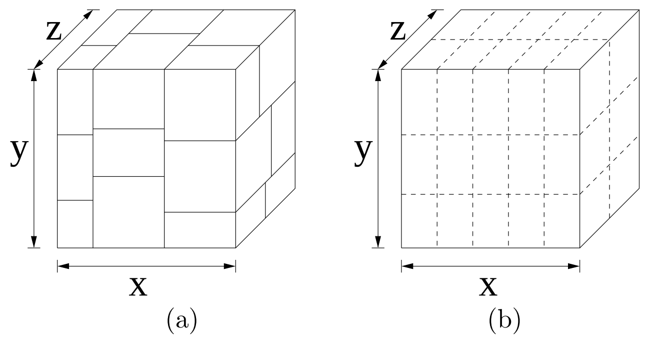


Figure 2: (a) Decomposition of total simulation space into 3*3*2 domains along x, y, z axises. (b) Division of total space into 5*3*2 equally sized cells.

The root data structure for each CPU (domain) is `home`. It is defined (in `Home.h`) as follows.

```
typedef struct _home Home_t ;
struct _home {
   int myDomain ;            /* index for this domain */
   int numDomains ;          /* total number of domains */
   int cycle ;               /* current simulation cycle */
   int subcycle ;            /* current subcycle within cycle */
   int lastCycle ;           /* cycle to stop simulation */

   Param_t *param;

   Node_t *nativeNodeQ ; /* head pointer of native node queue */
   Node_t *ghostNodeQ ;  /* head pointer of ghost node queue */
   ...
   Node_t **nodeKeys ;
   ...
   int *cellList ;
   int cellCount ;
   int nativeCellCount ;
   Cell_t **cellKeys ;

   int remoteDomainCount ; /* number of neighbor domains */
   int *remoteDomains ;    /* encoded indices of the neighbor domains */
   RemoteDomain_t **remoteDomainKeys ; /* pointers to RemoteDomain_t structs */
   real8 *domBoundX ;  /* array for the boundaries of all domains */
   real8 **domBoundY ;
   real8 ***domBoundZ ;

   real8 domXmin ;  /* boundary of this domain */
   real8 domXmax ;
   real8 domYmin ;
   real8 domYmax ;
   real8 domZmin ;
   real8 domZmax ;
} ;
```

`myDomain` specifies the the ID of the assigned domain for this CPU. The total number of domains is`numDomains`, which equals to `nXdoms`×`nYdoms`×`nZdoms`. The boundaries of all domains (as given by the domain decomposition) are stored in arrays `*domBoundX`, `**domBoundY`, `***domBoundZ`, while the boundary of the current domain is specified by `domXmin`, $\cdots$, `domZmax`. Pointer `*param` points a structure that holds all the control parameters of the simulation (see Section 2.4).

All the native nodes of the domain are stored in `**nodeKeys`, which is an array of pointers, each pointer points to the memory address of a `node` structure. The node tag specifies the entry of the node in the `**nodeKeys` array. For example, `nodeKeys[12]` in domain 2 stores the pointer of the node with tag `(2,12)`.

Each domain also holds an list of pointers to `remoteDomain` structures: they are (reduced) images of `home` structures on neighboring domains. The "RemoteDomain" structure is defined (in `RemoteDomain.h`) as follows.

```
typedef struct _remotedomain RemoteDomain_t ;
struct _remotedomain {
   int domainIdx ;   /* encoded index of this domain */
   int numExpCells ; /* number of native cells exported to this domain */
   int *expCells ;   /* list of encoded indices of the exprted cells */
   ...
   Node_t **nodeKeys ; /* indexed by node's tag.index, points to Node_t */
   ...
} ;
```

The `**nodeKeys` array of a `remoteDomain` stores the pointers of ghost nodes that are native to this `remoteDomain`. For example, if domain 0 has a copy of node $(1,24)$ (a native node of domain 1) as its ghost node, it can be referenced from domain 0 by `home->remoteDomains[1]->nodeKeys[24]`.

Looping through all native nodes by `nodeKeys[]` and all ghost nodes by `remoteDomains[]->nodeKeys[]` can be inconvenient. Thus all native nodes are linked to form a queue, with head pointer `*nativeNodeQ`. Similarly `*ghostNodeQ` is the queue head of all ghost nodes. The `*next` pointer in each node (see previous section) points to the next node in the queue.

1.3 Cells

For computational efficiency the simulation box is also divided into equally sized `cell`s, as shown in Fig. 2(b). The interaction between nodes in the same or neighboring cells are taken into account explicitly. The contributions from far away nodes are grouped into cells. Ideally, the number of nodes per domain should be around 20-100. The total number of cells should not exceed 1000.

The total number of cells for the entire problem is `nXcells`×`nYcells`×`nZcells`. The cells that are covered by or intersect with the current domain are called *native* cells. Cells that are not native but are neighbors of native cells are ghost cells. The array `**cellKeys` in `home` holds the pointers of all (native and ghost) `cell` structures. The `cell` structure is defined (in `Cell.h`) as follows.

```
typedef struct _cell Cell_t ;
struct _cell {
   Node_t  *nodeQ ;    /* queue head of nodes currently in this cell */
   int      nodeCount ;/* number of nodes on nodeQ */

   int     *nbrList ;  /* list of neighbor cell encoded indices */
   int      nbrCount ; /* number of neighbor cells */

   int     *domains ;  /* domains that intersect cell (encoded indices) */
   int      domCount ; /* number of intersecting domains */

   int     baseIdx ;   /* encoded index of corresp' base cell (-1 if not */
                       /* periodic)                                      */
   real8   xShift ;    /* if periodic, amount to shift corresp' base coord */
   real8   yShift ;
   real8   zShift ;
} ;
```

A cell could intersect multiple domains, the indices of which are stored in the `*domains` array, with total count `domCount`. All the nodes inside the cell are linked into a queue, headed by `*nodeQ`. The next node in this queue is pointed by `*nextInCell` in each `node`. The total number of nodes in this queue is `nodeCount`.

The indices of all cells that neighboring the current cell is stored in array `nbrList`, with total count `nbrCount`. When periodic boundary conditions (PBC) are used (default in DD3d), an extra layer of "virtual" cells are allocated around the simulation box. These "virtual" cells are periodic images of "real" cells within the simulation box. The `baseIdx` entry of a "real" cell is `-1`, while for a "virtual" cell it records the index of the corresponding "real" cell. The offset of the (base) "real" cell from the "virtual" cell is specified by (`xShift,yShift,zShift`).

Domains sharing the same cells or having cells neighboring each other are considered neighbors. In every cycle of DD3d simulation, domains exchange nodal information in these cells with their neighbors. For example, every `remoteDomain` contains a list of native cells `*expCells` of the current domain that needs to be exported to this `remoteDomain`.

Note: A second cell structure (`cell2`) is implemented to facilitate segment collision detection.

1.4 Parameters

The `*param` pointer in `home` structure points to a structure of all control parameters for DD3d simulation. Structure `param` is defined (in `Param.h`) as follows.

```
typedef enum {Periodic=0, Free=1, Reflecting=2} BoundType_t ;
typedef struct _param Param_t ;
struct _param {
   /* data decomposition */
   int nXcells, nYcells, nZcells ; /* numXcells, numYcells, numZcells */
   int nXdoms, nYdoms, nZdoms ;    /* numXdoms,  numYdoms,  numZdoms  */

   /* length and time scale */
   BoundType_t xBoundType ; /* Periodic (default), Free, or Reflecting */
   BoundType_t yBoundType ; /* Periodic (default), Free, or Reflecting */
   BoundType_t zBoundType ; /* Periodic (default), Free, or Reflecting */
   int   cycleStart ;       /* initial cycle number of simulation */
   int   maxstep;           /* maximum number of cycles of simulation */
   real8 timeStart ;        /* starting time of simulation */
   real8 timeNow;           /* current time */

   /* discretization control */
   real8 minSeg ;  /* min allowable segment length, before removing a node */
   real8 maxSeg;   /* max allowable segment length, before adding a node   */
   real8 velcutoff; /* upper bound of all segment velocities (sound barrier) */
   real8 rmax ;    /* maximum migration distance per timestep for any node */
   real8 rann ;    /* closest distance before dislocations are
                    * considered in contact */

   /* load control */
   int loadType ;  /* 0 Creep
                    * 1 Constant strain rate
                    * 2 Displacement-controlled
                    * 3 Load-controlled, load vs. time curve */
   real8 appliedStress[6]; /* applied stress when loadType = 0 */
   real8 eRate ;           /* strain rate    when loadType = 1 */
   real8 edotdir[3];       /* uniaxial loading axis companying eRate */

   /* materials property */
   real8 shearModulus, pois, YoungsModulus; /* elastic constants */
   real8 burgMag;      /* length unit */
   real8 MobScrew;     /* mobility of screw dislocation */
   real8 MobEdge;      /* mobility of edge dislocation */

   /* input/output control*/
   char dirname[300];  /* file directory name */
   int savecn,  savecnfreq,  savecncounter;
   int saveprop, savepropfreq, savepropdetail;
   char binfile[100] ; /* binary restart (input) file name */
   int binrestart ;    /* 1: write binary restart file (default) or 0: not */

   /* etc, etc */
   ...
} ;
```

Each entry in the `param` structure is bound to a string (in `ParseControl.C`) so that its value can be specified in input script files, such as `numXdoms = 4`. In text restart files, the values of all variable in the `param` structure written in the same format as in the input script file. Here we explain the meaning some most important control parameters. A more comprehensive explanation of control parameters and the format of input script files are given in Appendix A.

1. **Data decomposition**. `nXcells`, `nYcells`, `nZcells` specify the number of cells that divides the simulation box. `nXdoms`, `nYdoms`, `nZdoms` specifies the number of domains (CPUs) that divide for the simulation box.

2. **Boundary condition**. The boundary condition along x, y and z directions can be different. For example, `xBoundType`=0 means periodic boundary (PBC) along x direction, 1 means free boundary condition, 2 means reflecting boundary condition. So far only PBC is implemented (for all directions).

3. **Cycle and time**. The simulation will start at cycle `cycleStart`, which is probably the last cycle of a previous simulation. The simulation will continue for another `maxstep` cycles and end at cycle `cycleStart` + `maxstep`. The physical time at the beginning of simulation is `timeStart`. The current time during the simulation is `timeNow`. Note that in each simulation step (cycle) the increment of physical time is not necessarily the same.

4. **Discretization**. `minSeg` and `maxSeg` specifies how dislocation lines are meshed (in `Remesh.c`). If a node is connected with an arm with length smaller than `minSeg`, this node is removed, provided that it is a 2-node (node with 2 arms) and that both of its neighbors are 2-nodes. If an arm is longer than `maxSeg`, a new 2-node is inserted at the mid-point of the arm. The upper bound of node velocity is `velcutoff`. If a node velocity is higher than `velcutoff`, it is set to `velcutoff`. `rmax` specifies the maximum distance a node is allowed to travel within one simulation step. Let `vmax` be the maximum node velocity at a cycle (`vmax`≤`velcutoff`), the physical time increment of this step (`realdt`) must be smaller than `rmax`/`vmax` (=`deltaTT`).

5. **Load control**. DD3d can simulate several loading conditions. If `loadType`=0, creep condition is applied, i.e. a constant stress is applied. The 6-component stress tensor is specified in `appliedStress[6]` = $(\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{23}, \sigma_{31}, \sigma_{12})$ (in unit of Pa) and remains constant during the simulation. If `loadType`=1, DD simulation is performed at a constant strain rate given by `eRate` (in unit of 1/s). Currently only uniaxial tension/compression is implemented. The loading axis is given in Miller indices `edotdir[3]` = (h, k, l).

6. **Materials property**. DD3d assumes isotropic linear elasticity. Shear modulus μ is given by `shearModulus` (in unit of Pa); Poisson's ratio ν is given by `pois`. The Young's modulus Y (`YoungsModulus`) is computed by $Y = 2(1+\nu)\mu$. `shearModulus` and `pois` are used to compute elasticity driving forces on dislocation nodes (in `NodeForce.c`). `YoungsModulus` is used in computing the necessary loading stress in a constant strain rate loading simulation (in `LoadCurve.c`).

   All lengths and distances in DD3d are given in unit of `burgMag` (b), the fundamental lattice constant (or Burgers vector). `burgMag` itself is given in unit of meters. For example for BCC Mo we use `burgMag` = 2.725×10^{-10} (meter), i.e. $2.725\AA$. Time in DD3d is given in unit of second. Thus the velocity v has unit of b/s; dislocation mobility $M = v/(\tau b)$ has unit of $1/(Pa \cdot s)$. By default DD3d simulates a BCC crystal with anisotropic mobility for edge and screw dislocations, as specified by `MobEdge` and `MobScrew` respectively. A general dislocation with character angle θ (angle between Burgers vector $\vec{b}$ and line direction $\vec{u}$) has mobility `MobScrew` + (`MobEdge`−`MobScrew`)$\cdot\cos(\theta)$.

7. **Input/output**. At the beginning of the simulation, DD3d opens a subdirectory as specified by `dirname`, into which all output files are written. If `savecn`=1, a restart file `restart.cn` will be written at the end of the simulation. Moreover, intermediate restart files (e.g. `rs0000`, `rs0001`, etc.) are written every `savecnfreq` cycles. `savecncounter` sets the counter for the first intermediate restart files. For example, if `savecncounter`=20, the first restart file would be called `rs0020`, the next one would be `rs0021`, etc.

Property files would be saved at every `savepropfreq` cycles, if `saveprop`=1. `savepropdetail` specifies the levels of details (1-9) of the property files: 1 means a minimum amount of information, while 9 means many files will be saved.

If string `binfile` is not empty, DD3d will load a binary restart file (under the name specified by `binfile`) after the text restart file is loaded. If `binrestart`=1, each restart file DD3d creates will be accompanied by a binary restart file (using HDF5 library). If `binrestart`=0, no binary restart file will be created.

Note: **Moon**: please clean up `indxErate` (outdated)now that `edotdir` is in place.

Note: Clean up `cutoff`.

Note: Clean up `totalStrain`, `crystal` replaced by mobility law, `nrotload`, `mobility`, `mobilityFlag`, `mobilityRatio`, `poleOrder`, `strain[6]`, `edgeStrain[6]`, `screwStrain[6]`, `trueStrain`, `trueStrainED`, `trueStrainSC`, `totalStress`, `dbt`.

Note: Implement BCC mobility law based on kink energies and temperature.

Note: The previous entries `nXcells`, `nXdoms` etc in `Home` are now included in the `*param` structure. To make cell structure compatible with periodic boundary conditions (PBC), `nXcells`, `nYcells`, `nZcells` have to be all greater than or equal to 3.

Note: **Tim**: please clean up `fixed` in `Divide.c`. Use `minSeg` instead?

Note: Convert `nXcells` to `numXcells`. Convert `nXdoms` to `numXdoms`.

2 Algorithm Flow Chart

The main routine for DD3d simulation `AAmain.c` is reproduced below.

```
main (int argc, char *argv[]) {
   Home_t *home ;
   int cycleEnd;

   DD3dInit(argc,argv,&home);

   home->cycle = home->param->cycleStart ;
   home->subcycle = 0 ;
   cycleEnd = home->param->cycleStart + home->param->maxstep ;
   while (home->cycle < cycleEnd) {
       DD3dStep (home);
   }/* end of main loop */

   DD3dFinish (home);
   exit (0) ;
}
```

The structure is very simple. It begins with initialization by `DD3dInit()`. It then execute `DDStep()` for `maxstep` number of cycles. It finalizes by calling `DD3dFinish()`.

2.1 Initialization

Function `DD3dInit()` is not too complicated either. It calls several more functions to do its job.

```
void DD3dInit(int argc, char *argv[], Home_t **homeptr) {
    Home_t *home;

    home = InitHome () ;
    ...
    MPI_Init (&argc, &argv) ;
    MPI_Comm_rank (MPI_COMM_WORLD, &home->myDomain) ;
    MPI_Comm_size (MPI_COMM_WORLD, &home->numDomains) ;
    ...
    Initialize (home,argc,argv) ;
    ...
}
```

Function `InitHome` (in `InitHome.c`) merely allocates memory for the pointer `*home` and set the nodal queue heads to `NULL`. Afterwards, the `MPI` calls tell each CPU the index of its current domain (`home->myDomain`) and the total number of domains (`home->numDomains`).

Function `Initialize` then does the real job of setting up the simulation — reading in input files, decomposing data into domains, distributing data to all processors, and initializing other structures within `*home`, such as `cells` and `remoteDomains`. Text file input and domain decomposition are handled by CPU 0. After the data is distributed to other processors, all operations are then performed in parallel. Function `Initialize` is listed below.

```
void Initialize (Home_t *home,int argc, char *argv[]) {
   if (home->myDomain == 0) { /* only CPU 0 read text input file */
      ...
      for(i=1;i<argc;i++)
         ParseControl (argv[i], inData); /* read input files */
      ...
      /* decompose data only if binfile is not specified */
      if ((int)strlen(param->binfile) == 0) {
         ...
         if (param->repartition)       /* re-decompose data */
            Decomp (home, inData, &dComp) ;
         else /* decompose data keeping old domain boundary */
            OldDecomp (home, inData, &dComp) ;
         ...
      }
   }
   ...
   /* broadcast the params to all processors */
   MPI_Bcast ((char *)param, sizeof(Param_t), MPI_CHAR, 0, MPI_COMM_WORLD);

   if ((int)strlen(param->binfile) > 0) {
      /* if binfile exist, read binary file */
      ReadBin (home, param->binfile) ;
   } else {
      /* otherwise send decomposed data to all domains */
      InitSendDomains (home, inData, dComp, &intBuf, &intBufLen, &real8Buf,
                                            &real8BufLen) ;
      InitUnpackDomain (intBuf, intBufLen, real8Buf, real8BufLen, home) ;
      ...
   }
   InitCellNatives (home) ;   /* find native cells to this domain */
   InitCellNeighbors (home) ; /* find neighboring cells for each cell */
   InitCellDomains (home) ;   /* find domains intersecting cells */
   InitRemoteDomains (home) ; /* find domains neighboring this domain */
   ...
   OpenDir(home->param->dirname); /* open subdirectory for file output */

   return ;
}
```

This function is a bit long but quite easy to understand. First it reads in all the text input script files. They are passed as command line arguments when running DD3d. For example, if we run DD3d by `./dd3d a.script b.script`, then DD3d will read `a.script` and `b.script` one after the other. If no script file is specified, DD3d will read `control.script` by default. The format of text script files is always `varname = value`, such as `numXdoms = 3` and `MobScrew = 1`, etc. Each assignment statement is separated from others either by white spaces or a new line. If there are two assignment lines for the same variable, such as `numXdoms = 3` followed by `numXdoms = 4`, the later line overwrites the earlier line. The lines in a later script file overwrites those in the earlier script file. In this example, `numXdoms = 4` will overwrite `numXdoms = 3`.

After all text script files are read in, DD3d checks whether the string `binfile` is set. If it is, each domain will read in its nodal data from the binary file specified by `binfile`. If not, CPU 0 will decompose the nodal data it read in from the text file into domains and send them over to all the other domains. Each domain will receive the information on the nodes within their spatial limit.

Afterwards, each domain sets up their `cell` structures and `remoteDomain` structures, and the interconnectivity between them. Finally, DD3d opens a subdirectory specified by `dirname` and direct all future output file to this subdirectory.

2.2 Main Loop

After initialization, DD3d proceeds by executing `DD3dStep()` by a specified number (`maxstep`) of steps. Function `DD3dStep()` is outlined below.

```
void DD3dStep (Home_t *home) {
   ...
   /* Sort the native nodes into their proper cells */
   SortNativeNodes (home) ;
   ...
   /* Communicate ghost cells to/from neighbor domains */
   CommSendGhosts (home) ;
   ...
   /* Calculate the net charge tensor for each cell
    * involving a global communication */
   CellCharge (home) ;
   ...
   /* Calculate force on each native node */
   NodeForce (home);
   ...
   /* Calculate velocity for each native node */
   NodeVelocity (home);
   ...
   /* Update velocities of ghost nodes */
   CommSendVelocity (home) ;
   ...
   /* Sort nodes into cell2 structures for collision detection */
   SortNodesForCollision(home);
   ...
   /* Computes time step realdt, move nodes to the next time step
    * handles dislocation collision/reaction in between */
   AdvanceAndReconnect (home);
   ...
   /* Add and delete nodes to maintain a good mesh */
   Remesh (home) ;
   ...
   /* Send any nodes that have migrated out of the domain's
    * boundaries to the domain the node now belongs to.  */
   Migrate (home) ;
   ...
   /* Adjust applied stress according to loading condition */
   LoadCurve(home);
   ...
   /* Write output files */
   WriteStep (home) ;
   return ;
}
```

The `DD3dStep()` function specifies the main flow of the program. Every time `DD3dStep()` is called, it goes through 12 sub-steps and performs one computational cycle. We will briefly describes these 12 sub-steps in the following. Some of the sub-steps, such as `NodeForce`, `NodeVelocity`, `AdvanceAndReconnect`, `Remesh`, `Migrate`, `LoadCurve` will be described in more detail in subsequent chapters.

1. All the native nodes in the domain is queued to different cells according to their spatial distributions by `SortNativeNodes()`. Each `cell` has a pointer `*nodeQ` which is the queue head. Each node has a pointer `*nextInCell` that points to the next node in queue.

2. Each domain receives from its neighboring domains their native nodes if these nodes belongs to a cell that are neighbors of the current domain. The communication is handled by `CommSendGhosts()`. After that, these (ghost) nodes can be accessed from the `nodeKeys` array of the `remoteDomain` structures.

3. By calling `CellCharge()`, each domain obtain the *total* charge (tensor formed by Burgers vector and line direction) from all dislocation segments within the cell. This involves a global communication and it is not necessarily called every time `DD3dStep()` is called. We can do so if we assume that effects from far away segments fluctuate less than nearby segments in neighboring cell.

4. `NodeForce()` computes the total driving force on every native node of the domain. For each node, nodal contributions from 27 surrounding cells are explicitly computed. Contributions from nodes further away lumped into total cells charges.

5. `NodeVelocity()` computes the velocity of each native node based on its driving force, using the default or user-supplied mobility law subroutine. This routine is materials specific.

6. `CommSendVelocity()` send native node velocity to neighboring domains. After that, each domain also knows the velocity of their ghost nodes.

7. `SortNodesForCollision()` then queue all the nodes in every domain to different `cell2` structures, similar to `SortNativeNodes()` in step 1. This prepares for the next step.

8. At this point, every domain would need to move their nodes by a distance that equals to their velocity times a time step. In the easiest case, the time step `realdt` equals to `deltaTT` = `rmax` / `vmax`, where `rmax` is the maximum distance a node is allowed to move per time step, and `vmax` is the maximum node velocity. However, if dislocation segments collide within `deltaTT`, then the real time step `realdt` may need to become smaller. `AdvanceAndReconnect()` first predicts whether or not dislocation segments will collide with each other during period $[0, deltaTT]$. If they do, it tries to handle as many of them as possible. If `AdvanceAndReconnect` fails to handle all the events, the time step will be adjusted to the time of first event it fails to handle.

9. `Remesh()` adjusts the discretization of dislocation lines. If a node is connected with an arm with length smaller than `minSeg`, this node is removed, provided that it is a 2-node (node with 2 arms) and that both of its neighbors are 2-nodes. If an arm is longer than `maxSeg`, a new 2-node is inserted at the mid-point of the arm. Neighboring domains will do the same adjustments to their ghost nodes (after communication) to maintain consistency.

10. After advancing all the nodes according to their velocities, some nodes may go out of the boundary of its original native domain and become native to another domain. `Migrate()` sends these nodes to the new domain and the original native domain will mark these nodes as ghosts.

11. `LoadCurve()` adjusts the loading stress if the applied loading condition is not creep. For example, if constant strain rate is applied, the applied stress will be the difference between the applied strai and the current plastic strain, multiply the elastic constants.

12. `WriteStep()` writes property files such as dislocation density, stress-strain data at specified frequency during the simulation. It also writes restart files periodically.

3 Node Force

There are many ways to represent a continuous line in a computer: different choices in doing so correspond to different approximations being made. A certain choice may be suited for a certain type of applications, but it might simply reflect the personal taste of the modeler as well.

A simple way is to represent dislocations as a set of "nodes" interconnected by straight arms, as shown in Fig. 3, each arm representing a dislocation segment. While the nodes can move, each arm is associated with a Burgers vector that remains constant, until the destruction of the arm due to dislocation reaction. The definition of dislocation Burgers vector depends on the choice of line direction. For example, let $\vec{b}_{01}$ be the Burgers vector of the arm going from node 0 to node 1, and let $\vec{b}_{10}$ be the Burgers vector of the same arm going in the reverse direction, we have $\vec{b}_{01} + \vec{b}_{10} = 0$. In this way, the total Burgers vector of all arms going out of any given node is zero, e.g. $\vec{b}_{01} + \vec{b}_{02} + \vec{b}_{03} = 0$.

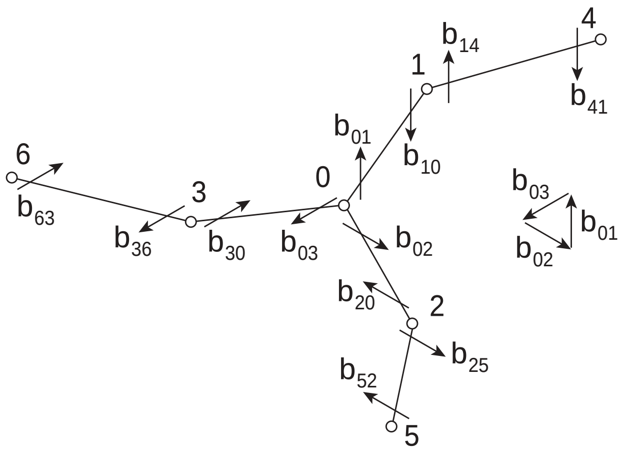


Figure 3: Dislocation network represented as a set of "nodes" (empty circles) interconnected by straight segments (see text).

Under this convention, a dislocation can be uniquely specified by a set of nodes $\{\mathcal{N}_i\}$, each described by its location $\vec{r}_i$ and connectivity with other nodes, i.e. $\mathcal{N}_i = \left[\vec{r}_i;\ \vec{b}_{ij},\ (j = 1, \cdots, n_i)\right]$, where n_i is the connectivity (number of arms) of node i. Let $E(\{\mathcal{N}_i\})$ be the total elastic energy of dislocation network. The driving force on node i can be rigorously defined as,

$$\vec{f}_i = -\frac{\partial E(\{\mathcal{N}_i\})}{\partial \vec{r}_i} \tag{1}$$

In other words, the driving force $\vec{f}_i$ equals to the rate of total energy drop $(-dE/d\vec{r}_i)$ of the dislocation network in response to an infinitesimal (virtual) displacement $(d\vec{r}_i)$ of node i, while keeping the topology (Burgers vectors of all arms) and positions of all other nodes constant.

The total elastic energy can be written as the sum of the self energies of all dislocation segments, and the mutual interaction energies between each segment pairs, such as

$$\begin{aligned} E(\{\mathcal{N}_i\}) &= W_S(01) + W_S(14) + W_S(02) + \cdots \\ &\quad + W_I(01,14) + W_I(01,02) + W_I(02,25) + \cdots \\ &= \sum_{\langle i,j\rangle} W_S(ij) + \frac{1}{2} \sum_{\langle i,j\rangle;\langle k,l\rangle} W_I(ij,kl)\ , \end{aligned} \tag{2}$$

where $W_S(ij)$ is the self energy of segment $\langle i,j\rangle$, $W_I(ij,kl)$ is the interaction energy between segments $\langle i,j\rangle$ and $\langle k,l\rangle$. In isotropic linear elasticity, analytical formula have been derived for both $W_S(i,j)$ and $W_I(ij,kl)$, for arbitrary segment orientation and Burgers vectors. Therefore, at least in principle, the driving force on node i can be evaluated by summing up the derivatives of every term in Eq. (2) with respect to $\vec{r}_i$. Of course, only those terms that involve node i will make non-zero contribution.

For example, let $\vec{L} = \vec{r}_1 - \vec{r}_0$ ($L = |\vec{L}|$) and $\vec{b} = \vec{b}_{01}$ ($b = |\vec{b}|$), the self energy of segment $\langle 0, 1\rangle$ is,

$$W_S(01) = W_S(\vec{L}, \vec{b}) = \frac{\mu}{4\pi}\left(b_s^2 + \frac{b_e^2}{1-\nu}\right) L \ln \frac{L}{e r_c} \tag{3}$$

where b_s and b_e are screw and edge components of the Burgers vector: $b_s = |\vec{b}\cdot\vec{L}|/L$, $b_e = (b^2 - b_s^2)^{1/2}$. μ is the shear modulus and ν is the Poisson's ratio of the material. r_c is the cut-off radius for the dislocation core. It is an undetermined parameter within the elasticity theory of dislocations. It can be pinned down by equating the total elastic energy of a given dislocation configuration (which depends on r_c) with the total energy of the same configuration in an atomistic simulation (see Chapter 3). r_c is typically on the order of $0.1b$.

The driving force contribution on node 1 due to the self energy of segment $\langle 0, 1\rangle$ is,

$$-\frac{\partial W_S(01)}{\partial \vec{r}_1} = \hat{\mathbf{e}}_L \cdot \left[-\frac{\mu}{4\pi}\left(b_s^2 + \frac{b_e^2}{1-\nu}\right)\ln\frac{L}{r_c}\right] \tag{4}$$

$$+ \hat{\mathbf{e}}_\theta \cdot \left[\frac{\mu}{4\pi}\left(\frac{2b_s b_e \nu}{1-\nu}\right)\ln\frac{L}{e r_c}\right] \tag{5}$$

where $\hat{\mathbf{e}}_L = \vec{L}/L$ is the unit vector along the segment direction, and $\hat{\mathbf{e}}_\theta \parallel \vec{b} - (\vec{b}\cdot\hat{\mathbf{e}}_L)\hat{\mathbf{e}}_L$ is the other unit vector on the plane containing both $\vec{L}$ and $\vec{b}$.

Hence we have obtained the self energy contributions to the nodal driving force. On the other hand, the interaction energy W_I between two generally orientated dislocation segments with arbitrary Burgers vectors is very complicated. For two general dislocation lines $C^{(1)}$ and $C^{(2)}$, with Burgers vectors $\vec{b}^{(1)}$ and $\vec{b}^{(2)}$, their interaction energy can be expressed as,

$$\begin{aligned} W_{12} &= -\frac{\mu}{2\pi}\int_{C^{(1)}}\int_{C^{(2)}} \frac{(\vec{b}^{(1)}\times\vec{b}^{(2)})\cdot(d\vec{l}^{(1)}\times d\vec{l}^{(2)})}{R} \\ &+ \frac{\mu}{4\pi}\int_{C^{(1)}}\int_{C^{(2)}} \frac{(\vec{b}^{(1)}\cdot d\vec{l}^{(1)})(\vec{b}^{(2)}\cdot d\vec{l}^{(2)})}{R} \\ &+ \frac{\mu}{4\pi(1-\nu)}\int_{C^{(1)}}\int_{C^{(2)}} (\vec{b}^{(1)}\times d\vec{l}^{(1)})\cdot T\cdot(\vec{b}^{(2)}\times d\vec{l}^{(2)}) \end{aligned} \tag{6}$$

where $R = \left(\sum_{i=1}^{3}(x_i^{(1)} - x_i^{(2)})^2\right)^{1/2}$, $T_{ij} = \partial^2 R/\partial x_i^{(2)}\partial x_j^{(2)}$. When $C^{(1)}$ and $C^{(2)}$ are straight lines, analytic formula can be derived by performing the integrals. In principle, one can then differentiate W_{12} with respect to $\vec{r}_1$ by brute-force. However, the analytic form of $\vec{f}_1$ thus obtained will be very complicated and numerically unstable.

Intuitively, the "local stress" (σ_{loc}) at the dislocation segment provides the driving force for dislocation motion, according to the Peach-Koehler formula $\vec{f} = (\vec{b}\cdot\sigma_{loc})\times\vec{L}$. The "local stress" is the superposition of the externally applied stress σ_{ext} and the internal stress produced by all other dislocation segments. Since the stress formula for arbitrary dislocation segments is easy to implement, most of the existing DD codes computes driving forces by computing this "local stress". Unfortunately, there has been much controversy and inconsistency in this type of approaches: because the stress field of a dislocation segment diverges as one approaches the segment itself, a neighboring segment would contribute to an infinitely large driving force to the segment of study. Besides, it is not clear why the self-energy contribution [Eq. (5)] should be arbitrarily ignored. To make the total driving force finite, it has been common practice to truncate the neighboring segments with an *ad hoc* truncation parameter. As we will see, all of these are unnecessary. Based on Eq. (1), the nodal driving force is well defined and stays finite, and can be computed without ambiguity. Below we will show the connection between the nodal driving force and the "local stress" field.

Consider the case where both lines $C^{(1)}$ and $C^{(2)}$ form complete loops. In this case, W_{12} equals to the integration of the stress field of loop 2 ($\sigma^{(2)}$) over the area of loop 1 ($A^{(1)}$), times the Burgers vector of loop

1 ($b^{(1)}$), i.e.

$$W_{12}^{\circ} = \int_{A^{(1)}} dA_{\beta}^{(1)} b_{\alpha}^{(1)} \sigma_{\alpha\beta}^{(2)} \tag{7}$$

Here we use W_{12}° (instead of W_{12}) to indicate the requirement that both $C^{(1)}$ and $C^{(2)}$ should be complete loops for Eq. (7) to be valid. The stress field from dislocation $C^{(2)}$ is given by

$$\begin{aligned}
\sigma_{\alpha\beta}^{(2)} &= -\frac{\mu}{8\pi} \oint_{C^{(2)}} b_m^{(2)} \epsilon_{im\alpha} \frac{\partial}{\partial x_i^{(2)}} \nabla^2 R dx_{\beta}^{(2)} \\
&- \frac{\mu}{8\pi} \oint_{C^{(2)}} b_m^{(2)} \epsilon_{im\beta} \frac{\partial}{\partial x_i^{(2)}} \nabla^2 R dx_{\alpha}^{(2)} \\
&- \frac{\mu}{4\pi(1-\nu)} \oint_{C^{(2)}} b_m^{(2)} \epsilon_{imk} \left(\frac{\partial^3 R}{\partial x_i^{(2)} \partial x_{\alpha}^{(2)} \partial x_{\beta}^{(2)}} - \delta_{\alpha\beta} \frac{\partial}{\partial x_i^{(2)}} \nabla^2 R \right) dx_k^{(2)}
\end{aligned} \tag{8}$$

where $\nabla^2 = \sum_{i=1}^{3} \partial^2 / \partial x_i^{(2)} \partial x_i^{(2)}$. However, if $C^{(2)}$ is not a complete loop, W_{12} no longer satisfies Eq. (7). Instead, one can show that

$$W_{12} = W_{12}^{\circ} + W_{12}^{\dagger} \tag{9}$$

$$W_{12}^{\dagger} \equiv -\frac{\mu}{4\pi} \int_{A^{(1)}} d\vec{A}^{(1)} \cdot (\vec{b}^{(1)} \times \vec{b}^{(2)}) \cdot \left(\frac{1}{|\vec{r}^{(1)} - \vec{r}_b|} - \frac{1}{|\vec{r}^{(1)} - \vec{r}_a|} \right) \tag{10}$$

where $\vec{r}_a$ and $\vec{r}_b$ are the starting and ending point of curve $C^{(2)}$, respectively.

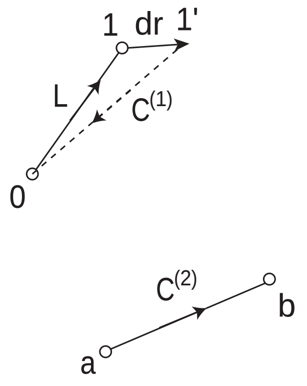


Figure 4: Computing the driving force on node 1 due to the interaction between segments $\langle 0, 1 \rangle$ and $\langle a, b \rangle$.

Let us now compute the nodal driving force contribution from the interaction between two straight dislocation segments, $\langle 0, 1 \rangle$ and $\langle a, b \rangle$, as shown in Fig. 4. Consider a virtual move of node 1 from $\vec{r}_1$ to $\vec{r'}_1 = \vec{r}_1 + d\vec{r}$, and consider the complete loop thus formed: $C^{(1)} = (0 \to 1 \to 1' \to 0)$. Let $C^{(2)} = \langle a, b \rangle$. The interaction energy between $C^{(1)}$ and $C^{(2)}$ can be described by Eq. (9),

$$W_{12} = \int_{A^{(1)}} d\vec{A}^{(1)} \cdot (\vec{b}^{(1)} \cdot \sigma^{(2)}) + W_{12}^{\dagger} \tag{11}$$

At the same time, the interaction energy between $C^{(1)}$ and $C^{(2)}$ can be decomposed into segment-segment interactions,

$$W_{12} = W_I(01, ab) + W_I(11', ab) + W_I(1'0, ab) \tag{12}$$

Therefore, the driving force on node 1 is

$$\begin{aligned}\vec{f}_1 &= -(W_I(01',ab) - W(01,ab))/d\vec{r} \\ &= W_{12}/d\vec{r} - W_I(11',ab)/d\vec{r} \\ &= W^{\circ}_{12}/dr + W^{\dagger}_{12}/d\vec{r} - W_I(11',ab)/d\vec{r}\end{aligned}$$

We can separate $\vec{f}_1$ into several parts,

$$\vec{f}_1 = \vec{f}^{\sigma}_1 + \vec{f}^{\rm corr}_1 \quad (13)$$

$$\vec{f}^{\sigma}_1 \equiv W^{\circ}_{12}/d\vec{r} \quad (14)$$

$$\vec{f}^{\rm corr}_1 \equiv \vec{f}^{\rm corr1}_1 + \vec{f}^{\rm corr2}_1 \quad (15)$$

$$\vec{f}^{\rm corr1}_1 = W^{\dagger}_{12}/d\vec{r} \quad (16)$$

$$\vec{f}^{\rm corr2}_1 = -W_I(11',ab)/d\vec{r} \quad (17)$$

Because the area $A^{(1)}$ enclosed by loop $C^{(1)}$ is a triangle, we have,

$$\int_{A^{(1)}} d\vec{A}^{(1)} = \int_0^L dx \frac{x}{L}(\hat{\mathbf{e}}_{01} \times d\vec{r}) \quad (18)$$

where $L = |\vec{L}|$, $\vec{L} = \vec{r}_1 - \vec{r}_0$, $\hat{\mathbf{e}}_{01} = \vec{L}/L$. Therefore,

$$\vec{f}^{\sigma}_1 = \int_0^L dx \frac{x}{L}\left[(\vec{b}^{(1)} \cdot \sigma^{(2)}(x)) \times \hat{\mathbf{e}}_{01}\right] \quad (19)$$

Therefore, the first term in Eq. (13) ($\vec{f}^{\sigma}_1$) accounts for the effect of "local stress", while the second term ($\vec{f}^{\rm corr}_1$) is a correction term (due to the fact that neither of the two dislocation segments are complete loops) previous not appreciated. Furthermore, $\vec{f}^{\sigma}_1$ is similar to, but not exactly the same as the Peach-Koehler formula ($\vec{f} = (\vec{b} \cdot \sigma_{loc}) \times \vec{L}$). Instead of being proportional to the total stress (due to $C^{(2)}$) along segment $\langle 0,1 \rangle$, $\vec{f}^{\sigma}_1$ is proportional to the weighted (by x) average of the stress field $\sigma^{(2)}(x)$ along the segment. In other words, it is proportional to the total torque that the stress field exerts on the segment $\langle 0,1 \rangle$ around node 0.

Based on Eq. (10), the first correction term is

$$\vec{f}^{\rm corr1}_1 = -\frac{\mu}{4\pi}\int_0^L dx \frac{x}{L}\left[(\vec{b}^{(1)} \times \vec{b}^{(2)}) \times \hat{\mathbf{e}}_{01}\right]\left(\frac{1}{|\vec{r}^{(1)} - \vec{r}_b|} - \frac{1}{|\vec{r}^{(1)} - \vec{r}_a|}\right) \quad (20)$$

To compute the second correction term $\vec{f}^{\rm corr2}_1$, we notice that $W_I(11',ab)$ is the interaction energy between a differential segment $d\vec{r}$ with a straight segment $\langle a,b \rangle$. From Eq. (6), we obtain

$$\begin{aligned}\vec{f}^{\rm corr2}_1 &= -W_I(11',ab)/d\vec{r} \\ &= \frac{\mu}{2\pi}\int_0^{L_2} dx_2 \frac{(\hat{\mathbf{e}}_{ab} \times (\vec{b}^{(1)} \times \vec{b}^{(2)}))}{R} \\ &- \frac{\mu}{4\pi}\int_0^{L_2} dx_2 \frac{\vec{b}^{(1)}\ (\vec{b}^{(2)} \cdot \hat{\mathbf{e}}_{ab})}{R} \\ &- \frac{\mu}{4\pi(1-\nu)}\int_0^{L_2} dx_2 (\nabla\nabla R \cdot (\vec{b}^{(2)} \times \hat{\mathbf{e}}_{ab})) \times \vec{b}^{(1)}\end{aligned} \quad (21)$$

where $L_2 = |\vec{L}_2|$, $\vec{L}_2 = \vec{r}_b - \vec{r}_a$, $\hat{\mathbf{e}}_{ab} = \vec{L}_2/L_2$, and $R = |\vec{r}_a + \hat{\mathbf{e}}_{ab}x_2 - \vec{r}_1|$.

Equations (19) (20) and (21) describe how to compute the nodal driving force due to the interaction between any two segment pairs. Below we will show that it also applies to the situation when the two segments share a common node: the stress field from each segment diverges at the common node, but the

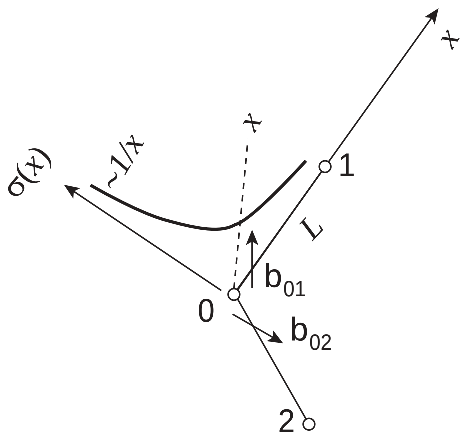


Figure 5: Interaction between two segments sharing a common node.

driving forces on all three nodes remain finite. As shown in Fig. 5, node 0 is connected with both node 1 and node 2. Let us examine the driving forces on all three nodes due to the interaction between $\langle 0,1\rangle$ and $\langle 0,2\rangle$.

Let $\sigma^{\langle 0,2\rangle}(x)$ be the stress field of segment $\langle 0,2\rangle$ on segment $\langle 0,1\rangle$. $\sigma^{\langle 0,2\rangle}(x)$ diverges as $x\to 0$ as $1/x$. However, $\vec{f}_1^{\sigma}$ remains finite, because

$$\vec{f}_1^{\sigma} \propto \int_0^L dx\cdot x\cdot \sigma^{\langle 0,2\rangle}(x) \tag{22}$$

The weight x cancels the singular term $1/x$ in $\sigma^{\langle 0,2\rangle}(x)$. One can easily check that $\vec{f}_1^{\rm corr}$ and hence $\vec{f}_1 = \vec{f}_1^{\sigma} + \vec{f}_1^{\rm corr}$ also remain finite. Similarly, $\vec{f}_2$ is finite too. Due to translational invariance, the driving force on node 0 is simply $\vec{f}_0 = -\vec{f}_1 - \vec{f}_2$.

Operationally, $\vec{f}_1^{\sigma}$ can be computed by numerical integration: we evaluate the stress field at several sampling points due to the other segment and compute their weighted average. On the other hand, analytical formula can be obtained for the correction term $\vec{f}_1^{\rm corr}$, by performing the integrals in Eq. (20) and (21). $\vec{f}_1^{\rm corr}$ can simply be computed as a function of $\vec{r}_0$, $\vec{r}_1$, $\vec{r}_a$ and $\vec{r}_b$.

We should notice that the correction terms tend to cancel each other once we sum up the different contributions to the driving force of a given node, say node 0. For brevity, let us call segments that are connected with node 0 *local* segments, and other segments *remote* segments.

1. The first correction term $\vec{f}_0^{\rm corr1}$ vanishes when we sum up the interactions between a *local* segment and *remote* segments that happen to form a closed loop.

2. The second correction term $\vec{f}_0^{\rm corr2}$ vanishes once we sum up the interactions between a remote segment with all local segments, because the Burgers vector of *local* segments sum to zero.

Therefore, when computing the driving force of node 0, an economic way is to first evaluate and sum up $\vec{f}_0^{\sigma}$ contributions from different segment pair interactions, and add the unbalanced correction terms at the end. There are two and only two places where unbalanced correction terms appear.

1. Let us first consider the interactions between *local* segments and *remote* segments. As stated above, when all these contributions are summed together, $\vec{f}_0^{\rm corr2} = 0$. However, $\vec{f}_0^{\rm corr1} \neq 0$, because the collection of all *remote* segments does not form a complete dislocation loop (assuming the entire dislocation network can be decomposed into completed loops, but now the local segments are excluded). We can make them a complete loop by adding auxiliary dislocations lines that goes from infinity to the neighbors of node 0 (opposite to the dashed lines in Fig. 6). Hence the sum of $\vec{f}_0^{\rm corr1}$ terms between all *remote* and all *local* segments equals to the sum of $\vec{f}_0^{\rm corr1}$ between the semi-infinite segments in Fig. 6 and all *local segments* of node 0. Notice that $\vec{f}_0^{\rm corr1}$ is non-zero only when two segments have

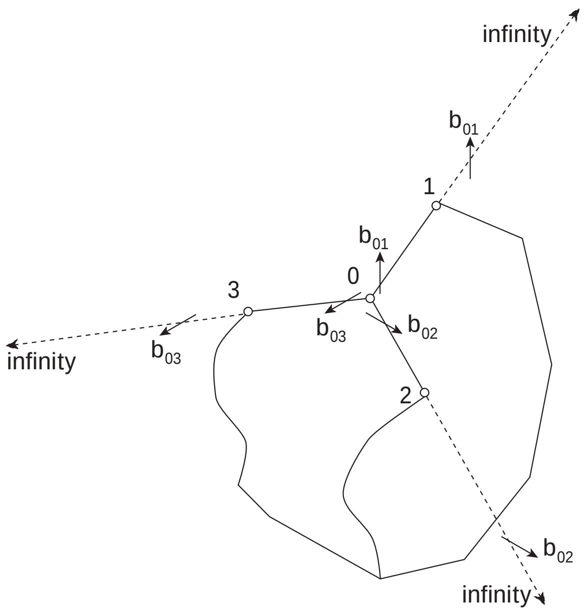


Figure 6: Cancellation of the correction terms $\vec{f}_0^{\rm corr1}$ by introducing auxiliary semi-infinite dislocation lines (see text).

different Burgers vectors. Therefore this correction is zero for all nodes with two arms. We only need to compute $\vec{f}_0^{\rm corr1}$ for nodes with three or more neighbors.

2. When computing the contributions from interactions between *local* segments, node 0 becomes the common node for every pair of *local* segments. In order to use the translational invariance property $\vec{f}_0 = -\vec{f}_1 - \vec{f}_2$, $\vec{f}_i$ $(i = 1, 2)$ should contain both $\vec{f}_i^{\rm corr1}$ and $\vec{f}_i^{\rm corr2}$.

The analytic form of stress field $\sigma^{(2)}(x)$ of a general straight dislocation segment with arbitrary Burgers vector (from which we compute $\vec{f}_1^{\sigma}$ by numerically integrating Eq. (19)) is available from standard text books of linear elasticity. Below we give the analytic forms of the correction terms $\vec{f}_1^{\rm corr1}$ and $\vec{f}_1^{\rm corr2}$.

From Eq. (20), $\vec{f}_1^{\rm corr1}$ can be written as,

$$\vec{f}_1^{\rm corr1}(\vec{r}_0, \vec{r}_1; \vec{r}_a, \vec{r}_b) = \vec{f}_1^{\rm corr1}(\vec{r}_0, \vec{r}_1; \vec{r}_b) - \vec{f}_1^{\rm corr1}(\vec{r}_0, \vec{r}_1; \vec{r}_a) \tag{23}$$

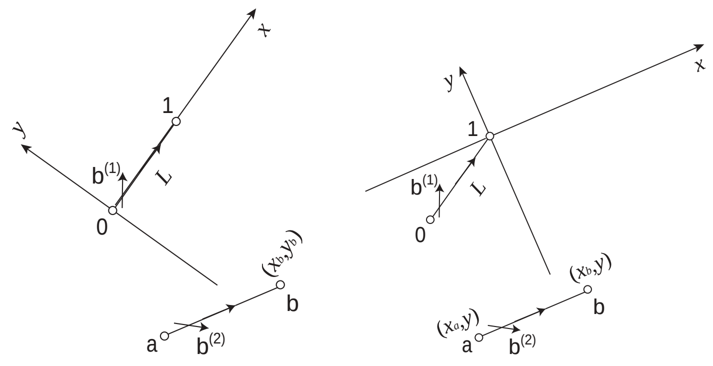
As shown in Fig. 7(a), we choose the coordinate system such that node 0 sits at origin, node 1 sits on the x axis, and node b sits on the x-y plane [at coordinate (x_b, y_b)]. Then,

$$\begin{aligned}\vec{f}_1^{\rm corr1}(\vec{r}_0, \vec{r}_1; \vec{r}_b) &= -\frac{\mu(\vec{b}^{(1)} \times \vec{b}^{(2)}) \times \hat{\mathbf{e}}_x}{4\pi L} \int_0^L \frac{x\,dx}{\sqrt{(x-x_b)^2 + y_b^2}} \\ &= -\frac{\mu(\vec{b}^{(1)} \times \vec{b}^{(2)}) \times \hat{\mathbf{e}}_x}{4\pi L} \left(R_{1b} - R_{0b} + x_b \ln \frac{R_{1b} + L - x_b}{R_{1b} - x_b} \right)\end{aligned} \tag{24}$$

where $R_{1b} = \sqrt{(L-x_b)^2 + y_b^2}$ ($R_{0b} = \sqrt{x_b^2 + y_b^2}$) is the distance between node 1 (node 0) with node b.

To compute $\vec{f}_1^{\rm corr2}$, we choose the coordinate system as shown in Fig. 7(b). Node 1 is at the origin. Node a and b lies in the x-y plane and along the x axis. Let the coordinates of node a and b be (x_a, y) and (x_b, y) respectively. Based on Eq. (21), $\vec{f}_1^{\rm corr2}$ can be written as,

$$\vec{f}_1^{\rm corr2} = \left[\frac{\mu}{2\pi}(\hat{\mathbf{e}}_x \times (\vec{b}^{(1)} \times \vec{b}^{(2)})) - \frac{\mu}{4\pi}(\vec{b}^{(2)} \cdot \hat{\mathbf{e}}_x)\vec{b}^{(1)}\right] \int_{x_a}^{x_b} \frac{dx}{\sqrt{x^2 + y^2}}$$



(a) (b)

Figure 7: Geometry for computing (a) $\vec{f}_1^{\,\mathrm{corr1}}$ and (b) $\vec{f}_1^{\,\mathrm{corr2}}$.

$$- \quad \frac{\mu}{4\pi(1-\nu)}\left[(\vec{b}^{(2)}\times\hat{\mathbf{e}}_x)\cdot\int_{x_a}^{x_b}dx\nabla\nabla\frac{1}{\sqrt{x^2+y^2+z^2}}\right]\times\vec{b}^{(1)} \tag{25}$$

We can make use of the following identities,

$$\int_{x_a}^{x_b}\frac{dx}{\sqrt{x^2+y^2}} \quad = \quad \ln\frac{R_{1b}+x_b}{R_{1a}+x_a} \quad \left(=\ln\frac{R_{1a}-x_a}{R_{1b}-x_b}\right)$$

where $R_{1b}=\sqrt{x_b^2+y^2}$ ($R_{1a}=\sqrt{x_a^2+y^2}$) is the distance between node b (node a) and node 1. Because $z=0$, we also have,

$$\nabla\nabla\frac{1}{\sqrt{x^2+y^2+z^2}} \quad = \quad \begin{bmatrix} \frac{1}{R}-\frac{x^2}{R^3} & \frac{-xy}{R^3} & \frac{-xz}{R^3} \\ \frac{-xy}{R^3} & \frac{1}{R}-\frac{y^2}{R^3} & \frac{-yz}{R^3} \\ \frac{-xz}{R^3} & \frac{-yz}{R^3} & \frac{1}{R}-\frac{-z^2}{R^3} \end{bmatrix} = \begin{bmatrix} \frac{y^2}{R^3} & \frac{-xy}{R^3} & 0 \\ \frac{-xy}{R^3} & \frac{x^2}{R^3} & 0 \\ 0 & 0 & \frac{1}{R} \end{bmatrix}$$

where $R=\sqrt{x^2+y^2}$. Different components of this matrix can be integrated out separately,

$$\begin{aligned} \int_{x_a}^{x_b}\frac{dx}{R^3} &= \left.\frac{x}{y^2R}\right|_{x_a}^{x_b} \\ \int_{x_a}^{x_b}\frac{xdx}{R^3} &= -\left.\frac{1}{R}\right|_{x_a}^{x_b} \\ \int_{x_a}^{x_b}\frac{x^2dx}{R^3} &= \left[\ln(R+x)-\frac{x}{R}\right]_{x_a}^{x_b} \end{aligned}$$

Therefore,

$$\begin{aligned} \vec{f}_1^{\,\mathrm{corr2}} &= \left[\frac{\mu}{2\pi}(\hat{\mathbf{e}}_x\times(\vec{b}^{(1)}\times\vec{b}^{(2)}))-\frac{\mu}{4\pi}(\vec{b}^{(2)}\cdot\hat{\mathbf{e}}_x)\vec{b}^{(1)}\right]\ln(R+x)\big|_{x_a}^{x_b} \\ &- \frac{\mu}{4\pi(1-\nu)}\left((\vec{b}^{(2)}\times\hat{\mathbf{e}}_x)\cdot\begin{bmatrix} x/R & y/R & 0 \\ y/R & \ln(R+x)-x/R & 0 \\ 0 & 0 & \ln(R+x) \end{bmatrix}_{x_a}^{x_b}\right)\times\vec{b}^{(1)} \end{aligned} \tag{26}$$

Please notice that in Eq. (24) and (26) the choices of coordinate system are different.

4 Mobility Laws

We have seen that the driving forces on dislocations can be rather complicated to compute, as compared with, say, atoms in Molecular Dynamics simulations. At the same time, the equation of motion for dislocation lines is by no means as simple as that for atoms either. The Newton's equation of motion for atoms ($\vec{f} = m\vec{a}$) is completed specified by a single parameter, the atomic mass m, once the driving force $\vec{f}$ is known. The equation of motion for dislocations, on the other hand, is much more complicated for several reasons. First, dislocations are line-like (1-dimensional) objects, as compared with atoms, which are point-like (0-dimensional) objects. Second, dislocation's response to driving forces can be very anisotropic with respect to the glide plane (the plane defined by the line direction and the Burgers vector): motion out of the glide plane (climb) is much more difficult than motion within the glide plane.

Under conventional loading conditions, dislocation motion can usually be regarded as being over-damped: the inertia effect can be safely ignored. In this case, the equation motion becomes first order: $\vec{v} = \mathcal{M}(\vec{f})$. The mobility function $\mathcal{M}$ specifies the instantaneous velocity $\vec{v}$ of dislocation in response to the driving force $\vec{f}$. While being first order is somewhat simpler than the (second order) Newton's equation of motion, the mobility function $\mathcal{M}$ however, can be quite complicated.

Model FCC Mobility Law

Let us start with the simplest possible mobility function that mimics the behavior of real dislocations. In this case, we assign every dislocation segment a glide plane, and confine its motion entirely within its glide plane. Because a discretization node is meant to be a sampling point on a dislocation line, its two arms must have the same glide plane, which is the plane on which the node itself is confined to move.

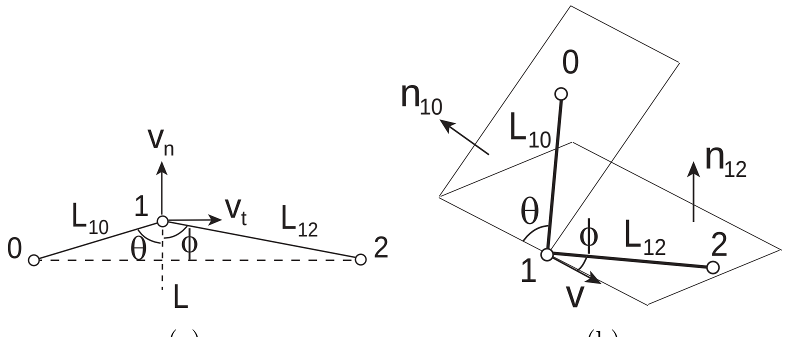


Figure 8: (a) Velocity of discretization node 1. The glide plane of both arms of node 1 ($\vec{n}_{10}$ and $\vec{n}_{12}$) are the same as the plane of the paper. (b) Velocity of physical node 1.

As shown in Fig. 8(a), there are two independent directions in this plane, and our mobility law needs to specify the node velocity in this two directions: v_n for motion normal to the "local tangent" and v_t for motion along the "local tangent". For simplicity, we can set $v_t \equiv 0$, because for a line only motion perpendicular to itself is physical. However, we can also assign non-zero values for v_t such that the meshing of the dislocation line is automatically adjusted (e.g. to maintain approximately equal segment lengths, see Problem 1).

Let f_n be the projection of nodal driving force $\vec{f_1}$ along the "normal" direction and L be the distance between node 0 and node 2. $f_n/(L/2)$ thus represents the average force (per unit length) on dislocation line around node 1. The simplest mobility law one can construct is to have dislocation velocity proportional to the force per unit length:

$$v_n = M f_n/(L/2) = 2M f_n/(L_{10}\sin\theta + L_{12}\sin\phi) \quad (27)$$

In DD simulations, we will encounter not only discretization nodes, but also physical nodes, which can be the place where a dislocation line bends sharply around a corner (Fig. 8(b)), or where three dislocation lines meet. An often encountered situation is that the arms of the physical node have different glide planes. In this case, the node velocity has to simultaneously satisfy several glide constraints. If there are two linearly independent glide plane normal directions, such as in Fig. 8(b), the nodal velocity has to be aligned along the intersection line between the two glide planes. Let f be the nodal force component along this direction. The magnitude of nodal velocity can be evaluated as,

$$v = 2Mf/(L_{10}\sin\theta + L_{12}\sin\phi) \tag{28}$$

(what if physical nodes able to move in 2- or even in 3 dimensions? A good remeshing is important, especially around a physical node: to have approximately equal arm lengths around a physical node.)

Implementation of the mobility law described above requires the knowledge the glide plane normal vector for every dislocation arm. This can be set in the initial condition and remain fixed during the simulation. When a new discretization node is inserted in an arm during remeshing, the two new segments will inherit the glide planes of the original segment. However, when new dislocation segments are created by dislocation reaction, additional rules must be supplied to specify their glide planes. Below we give an example of such rules.

1. Let $\vec{l}$ be the unit vector of the line direction of the dislocation segment and $\vec{b}$ be its Burgers vector. If $|\vec{l}\times\vec{b}| \geq \epsilon$, (e.g. $\epsilon = 0.001$) the glide plane normal is $\vec{n} = \vec{l}\times\vec{b}/|\vec{l}\times\vec{b}|$.

2. When $|\vec{l}\times\vec{b}| < \epsilon$, i.e. $\vec{l}$ and $\vec{b}$ are almost parallel, the above procedure is numerically unstable. This is a manifestation of the fact that a screw dislocation can principally move in multiple planes. Choose $\vec{n}$ randomly from a pre-specified table of glide plane normals $\{\vec{n}_i\}$ for screw dislocations with Burgers vector $\vec{b}$.

In last step above, the choice of $\vec{n}$ out of table $\{\vec{n}_i\}$ may not be completely random: the probability could be weighted by the projection of the nodal driving force on different candidate planes (see Problem 2).

In the mobility law above, every dislocation segment has a glide plane, even for screw dislocations. This mimics the dislocations in FCC metals, in which dislocations are dissociated into partials bounding an area of stacking fault on $\{111\}$ planes. Therefore, all dislocations (with Burgers vector $\frac{1}{2}\langle 110\rangle$) are confined to move on $\{111\}$ planes. On the other hand, screw dislocations can change its glide plane from one $\{111\}$ plane to another. Such rare cross-slip events are the result of thermal fluctuation and possibly a change of the nodal driving force (see Problem 2).

Model BCC Mobility Law

Let us now explore alternative choices of simple mobility functions. We have just considered an extreme case where even screw dislocations have a glide plane. We need to remind ourselves that the cause of this phenomena – planar dissociation of dislocations – occurs in FCC metals but is not generally true for all materials. For example, screw dislocations in BCC metals have a compact core and are found to move in arbitrary directions at high temperatures, both in experiments and in direct atomistic simulations (see Chapter 3). Such behavior is called "pencil-glide". In the following we explore the proper mathematical forms for the mobility function to describe pencil-glide.

Let us first consider the mobility function for a segment: $\vec{v} = \mathcal{M}(\vec{f})$. This applies to discretization nodes with nodal force $\vec{f_n}$: as shown in Fig. 8 $\vec{f_n}/(L/2)$ approximating the force per unit length is used in the above mobility function.

Here we completely abandon the notion of a glide plane. The mobility function is hence stateless: the driving force completely determines the dislocation velocity. More specifically, the dislocation velocity only depends on the relative direction of the dislocation segment w.r.t. its Burgers vector, and of course, the driving force.

As shown in Fig. 9, let θ be the angle between $\vec{b}$ and $\vec{l}$, and ϕ be the angle between the glide plane formed by $\vec{b}$ and $\vec{l}$ (normal vector $\vec{n}$) and a reference direction in the crystal. Therefore the mobility function can be written as $\vec{v} = \mathcal{M}(\vec{f},\theta,\phi)$. Furthermore, because motion along the dislocation line itself is unphysical, both $\vec{v}$ and $\vec{f}$ are two dimensional vectors in the tangent plane of the unit sphere, spanned by $\vec{n}$ and $\vec{t} = \vec{l}\times\vec{n}$.

For simplicity, we ignore the dependence on ϕ, i.e. we assume the mobility is isotropic w.r.t. the orientation of the glide plane. Dislocation mobility is then only a function of the character angle θ.

Define $f_n = \vec{f}\cdot\vec{n}$, $f_t = \vec{f}\cdot\vec{t}$, $v_n = \vec{v}\cdot\vec{n}$, $v_t = \vec{v}\cdot\vec{t}$. f_n and v_n are the climb force and velocity, i.e. for motion out of the glide plane, while f_t and v_t are glide force and velocity, i.e. for motion within the

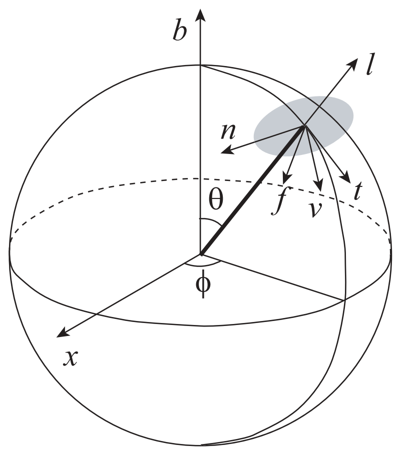
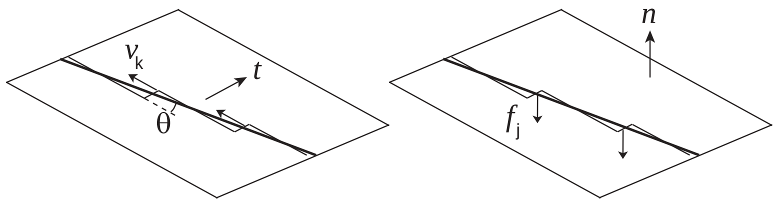


Figure 9: Generalized mobility function $\vec{v} = \mathcal{M}(\vec{f}, \theta, \phi)$. Both $\vec{v}$ and $\vec{f}$ are two dimensional vectors in the tangent (shaded) plane spanned by $\vec{n}$ and $\vec{t}$.



(a) (b)

Figure 10: Constructing generalized mobility function by considering the behavior of dislocations in the vicinity of screw orientation ($\theta \sim 0$).

glide plane. For edge dislocations ($\theta = 90°$) the dislocation mobility along these two directions are very different: $M_g/M_c \gg 1$, where $M_g = v_t/f_t$ and $M_c = v_n/f_n$. On the other hand, the mobility for screw dislocation is isotropic for pencil glide. In fact, when $\vec{b} \parallel \vec{l}$, we are no longer able to define vectors $\vec{n}$ and $\vec{t}$, in order to differentiate "glide" and "climb" directions. For screw dislocations ($\theta = 0°$), we assume $\vec{v} = M_s\vec{f}$. For all other orientations (mixed dislocations), the mobility function changes smoothly from that of edge to screw. Let us now construct the simplest mobility function of this kind, described by three parameters ($M_g > M_s > M_c$).

Consider mixed dislocations in the vicinity of screw orientation ($\theta \sim 0$). As shown in Fig. 10(a), it can be regarded as long screw segments connected by short edge segments. Each edge segment is an atomic sized kink with height h. Assume for the moment that the screw segments cannot move ($M_s = 0$), while the kinks move with velocity v_k. Let L be the average distance between kinks. If the kinks move laterally by L, the entire dislocation moves forward (along $\vec{t}$) by h. Thus the dislocation glide with velocity $v = v_k h/L = v_k \sin\theta$. Let $M_k = v_k/f_t$ be the mobility of kinks, then the glide mobility of the dislocation can be written as $M_t(\theta) = M_k \sin\theta$ (if $M_s = 0$). In general M_s is finite and both the screw and edge segments contribute to dislocation mobility, which can be written as,

$$M_t(\theta) \equiv v_t/f_t = (M_k^2 \sin^2\theta + M_s^2)^{1/2} \tag{29}$$

Now consider the motion of the dislocation perpendicular to the glide plane, i.e. along normal direction $\vec{n}$, as in Fig. 10(b). For the moment assume screw segments can move with zero resistance $1/M_s = 0$, while all the resistance comes from the climb motion of edge segments. In this orientation, the kinks in Fig. 10(a) are called "jogs". Let f_j be the drag force that each jog produces while moving with velocity v_n and $M_j = v_n/f_j$ be the jog mobility. The average drag force per unit length on the dislocation is $f_j \sin\theta$

which must be balanced by the climb driving force f_n. Therefore, the drag coefficient (inverse of mobility) is $1/M_n(\theta) = \sin\theta/M_j$. In general when $1/M_s$ is finite, both jog and screw segments produce drag forces during motion,

$$\begin{aligned} 1/M_n(\theta) \equiv f_n/v_n &= (\sin^2\theta/M_j^2 + 1/M_s^2)^{1/2} \\ M_n(\theta) &= (M_j^{-2}\sin^2\theta + M_s^{-2})^{-1/2} \end{aligned} \tag{30}$$

To ensure that when $\theta = 90^\circ$, $M_t(\theta) = M_g$ and $M_n(\theta) = M_c$, we have,

$$\begin{aligned} M_k &= (M_g^2 - M_s^2)^{1/2} \\ M_j &= (M_c^{-2} - M_s^{-2})^{-1/2} \end{aligned}$$

Therefore, the mobility function $\vec{v} = \mathcal{M}(\vec{f})$ can be written explicitly as,

$$\begin{aligned} \vec{v} &= \left[M_t(\theta)\vec{t}\otimes\vec{t} + M_n(\theta)\vec{n}\otimes\vec{n}\right]\vec{f} \\ &= M_t(\theta)\vec{f} - \left[M_t(\theta) - M_n(\theta)\right](\vec{n}\cdot\vec{f})\vec{n} \end{aligned} \tag{31}$$

Notice that $\vec{b}\times\vec{l} = \vec{n}\sin\theta$, we have

$$\begin{aligned} \vec{v} &= M_t(\theta)\vec{f} - \frac{M_t(\theta) - M_n(\theta)}{\sin^2\theta}\left[(\vec{b}\times\vec{l})\cdot\vec{f}\right](\vec{b}\times\vec{l}) \\ &\equiv M_t(\theta)\vec{f} - g(\theta)\left[(\vec{b}\times\vec{l})\cdot\vec{f}\right](\vec{b}\times\vec{l}) \end{aligned} \tag{32}$$

One can show that function $g(\theta)$ is well defined and continuous for all θ and $g(0) = \frac{1}{2}M_s\left[(M_k/M_s)^2 + (M_s/M_j)^2\right]^{1/2}$ (Problem 3).

Eq. (32) specifies the mobility of a generally orientated dislocation segment. This mobility function is completely specified by three parameters: M_g (edge glide mobility), M_c (edge climb mobility) and M_s (screw mobility). By setting $M_c \ll M_s < M_g$, we effectively confine the motion of mixed dislocations to move on the glide plane containing both $\vec{b}$ and $\vec{l}$.

Because motion of screw dislocation is completely unconstrained (it simply follows the driving force direction), this mobility law corresponds to the behavior of high temperature behavior of BCC metals where "pencil-glide" of screw dislocations are observed. At low temperatures, screw dislocation in BCC metals still prefer to move along crystallographic planes (e.g. $(1\bar{1}0)$ intersecting the $\frac{1}{2}\langle 111\rangle$ Burgers vector). planes, even though it is not dissociated (hence confined) in any plane. To describe such behavior, the mobility function must depend on ϕ in Fig. 9 as well (Problem 4).

The above mobility function can be directly used to compute velocity of discretization nodes. The input of the mobility function is force per unit length, which is nodal force divided by the average length of the two arms [$L/2$ in Fig. 8(a)]. For discretization nodes, the (could be more than three) arms of the node may have different mobility functions and have to be taken into account separately. Take the configuration in Fig. 8(b) as an example, the velocity of node 1 can be obtained by solving the following set of equations,

$$\begin{aligned} \vec{v} &= M_{01}(\vec{f}_{01}) \\ \vec{v} &= M_{12}(\vec{f}_{12}) \\ \vec{f} &= \vec{f}_{01}L_{01}/2 + \vec{f}_{12}L12/2 \end{aligned} \tag{33}$$

When the mobility functions have the tensor form as in Eq. (32), we can solve for $\vec{v}$ as

$$\vec{v} = \left(\frac{2}{L_{01}}M_{01}^{-1} + \frac{2}{L_{12}}M_{12}^{-1}\right)^{-1}\vec{f} \tag{34}$$

where $(\cdot)^{-1}$ corresponds to matrix inversion.

5 Event Detection and Handling

The previous section described how DD3d calculates the velocity (v_i) of every node at the beginning of each iteration. If one takes a fixed value of timestep (Δt) and advance each node by $v_i \cdot \Delta t$, it is the possible that some segments will cut through each other, which would be unphysical. A realistic simulation will need to detect the occurrence of such (contact) events and change the topology (line connection) of dislocations according to physical laws. This is handled by function `AdvanceAndReconnect`, as listed below,

```
void AdvanceAndReconnect (Home_t *home) {
  param = home->param;
  ...
  /* detect possible segmental collisions, arrange segment pairs
     (nodal quadrupoles) according to projected time */
  DetectAllEvents (home);

  /* select the earliest event among all domains */
  SelectEvent (home) ;

  /* move all the nodes by time param->realdt */
  AdvanceAllNodes (home) ;

  /* Calculate the plastic strain increment */
   DeltaPlasticStrain (home) ;
  ...
  /* execute the selected event */
  ExecuteEvent (home) ;
  return;
}
```

Function `DetectAllEvents` first predicts whether, when and where collisions will occur. The earliest event is then selected (by `SelectEvent`) whose time then replaces the timestep. `AdvanceAllNodes` then move all the nodes by this time, so that the predicted colliding segments will be just touching each other. Line reconnection is then handled by `ExecuteEvent`. This section will discuss the detection algorithm in DD3d. The topology handling (line reconnection) will be discussed in the next section.

5.1 Four Types of Events

If all nodes are moving with constant velocities, then four types of topological events could occur in the time window of $[0, \Delta t]$, as shown in Fig. 11. Type (1) is the most general case, in which two segments that previously were not in contact come into contact. We also call type (1) event a "collision" event. A type (2) event corresponds to two segments that shared a common node become overlap on each other, and we also call it a "zipping" event. In type (3), the length of a segment shrink down to zero, and we call it a "shrinking" event. A type (4) event describes a 4-node dissociates into two 2- or 3-nodes, and we call it a "dissociation" event. The current version of DD3d does not handle type (4) event. We will neglect it in our discussion from now on. `DetectAllEvents` calls four other functions to detect each of the four types of topological events.

```
void DetectAllEvents (Home_t *home) {
  ClearEventList (home) ;

  /* Type 1 events */
  DetectCollision (home) ;

  /* Type 2 events */
  DetectZipping (home) ;
```

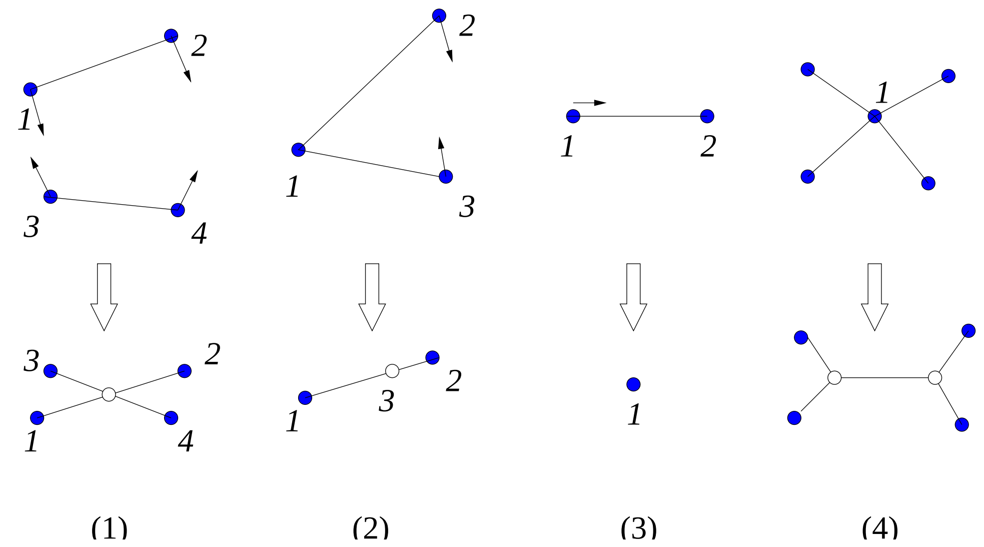


Figure 11: Four types of topological events possible in DD3d simulation.

```
  /* Type 3 events */
  DetectShrinking (home) ;

  /* Type 4 events */
  Detect4NodeDissociation (home) ;
}
```

For example, function `DetectCollision` loops through all segment pairs that are not connected with each other, and compute whether or not they will come in contact during $[0, \Delta t]$. Function `DetectZipping` loops through all nodes and check whether it has to arms that will become overlapped. `DetectShrinking` loops through all segments and check whether their length will shrink to zero, etc. Whenever an event is detected, an entry is added to the `EventList`, which is an array of the following structure, as specified by `Topology.h`.

```
struct _topevent {
    int type ;
    int dom1, idx1, dom2, idx2;
    int dom3, idx3, dom4, idx4;
    real8 tp ;
    real8 alpha, beta, x, y, z ;
} ;
```

The first entry of the `_topevent` structure specifies the event type (1-4). The next 8 integers specify the nodes involved in this event. `tp` is the predicted time of this event, $(0 \le t_p \le \Delta t)$. `alpha, beta, x, y, z` specifies the spatial location of this event. For example, in a type (1) event, the collision point $r = (x, y, z)$ is given by $r = (1 - \alpha) r_1 + \alpha r_2 = (1 - \beta) r_3 + \beta r_4$.

It turns out that only a general algorithm that can detect type (1) collision is needed, because type (2) and (3) events can be considered as degenerate cases of type (1). Similarly, the procedure that handles type (1) collision can be easily adapted to handle type (2) and (3) events, as will be discussed in Section 5.

It is a complex computational geometry problem to determine whether, when and where two line segments will come into contact, given arbitrary initial position and velocity of the four end nodes. The following subsection will discuss the algorithm used in DD3d to solve this problem.

5.2 Event Detection

5.3 Event Handling

Need to Write: `ExecuteEvent()` in `EventHandle.c`

```
void ExecuteEvent (Home_t *home) {
    domid = home->selectevdom ;
    evid  = home->selectevid ;
    ClearOpList (home);
    if(evid<0) return ; /* no event */
    ...
    if(type == 1)
    {/* Collision, type = 1 */
        ...
        if(...)
            if(...)
                HandleNodeNodeCollision (home,nodeA,nodeB);
            else
                HandleNodeSegCollision (home,nodeA,node3,node4);
        else
            if(...)
                HandleNodeSegCollision (home,nodeB,node1,node2);
            else
                HandleSegSegCollision (home,node1,node2,node3,node4);
    }
    else if(type == 2)
    {
        if( ... )
            HandleNodeNodeCollision (home, node2, node3);
        else if (...)
        {
            ...
            HandleNodeSegCollision (home, node2, node1, node3);
        }
        ...
    }
    else if(type == 3)
    {
        HandleNodeNodeCollision (home, node1, node2);
    }
    ...
}

void HandleSegSegCollision (Home_t *home, Node_t *node1,
                            Node_t *node2, Node_t *node3, Node_t *node4)
{
    ...
    node5 = InsertNewNode (home, node1, node2, x, y, z, 1) ;
    node6 = InsertNewNode (home, node3, node4, x, y, z, 1) ;
    HandleNodeNodeCollision (home, node5, node6);
}
```

```
void HandleNodeSegCollision (Home_t *home,
                             Node_t *node2, Node_t *node3, Node_t *node4)
{
    ...
    node6 = InsertNewNode (home, node3, node4, x, y, z, 1);
    HandleNodeNodeCollision (home, node2, node6);
}

void HandleNodeNodeCollision (Home_t *home, Node_t *node2, Node_t
*node4) {
    /* remove all arms from node2 to node4 */
    ChangeArmBurgID (home, node2, node4, -1, 1);

    /*  Redirect all connections with node4 to node2, delete node4 */
    for(j=0;j<node4->numNbrs;j++)
    {
        ...
        ChangeConnection (home, nbr, node4, node2, 1);
        InsertArm (home, node2, nbr, bid, 0, 0, 0, 1);
    }
    RemoveNode (home, node4, 1) ;

    /* Check whether node2 has arms linking to the same neighbor
     *  combine these arms */
    for(j=0;j<node2->numNbrs;j++)
    {
        ...
            ChangeArmBurgID (home, node2, nbr, -1, 1);
            ChangeArmBurgID (home, nbr, node2, -1, 1);
        ...
            /* check whether nbr node becomes isolated due to arm
               removal, if yes then remove this node */
            if (NodeConnectivity(nbr)==0) RemoveNode (home, nbr, 1);
        ...
    }
    /* Check whether node2 become isolated nodes, if yes remove this node */
    if (NodeConnectivity(node2)==0) RemoveNode (home, node2, 1);
    return;
}
```

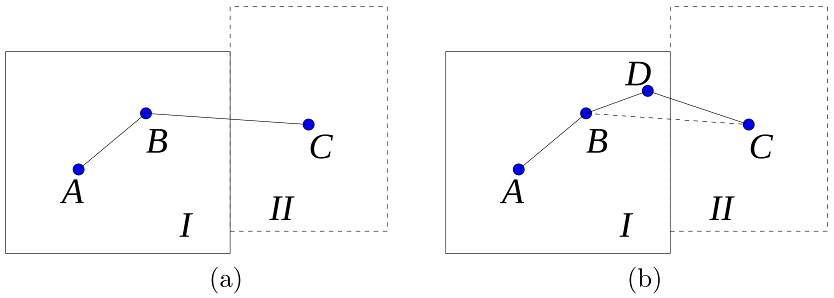


Figure 12: (a) Nodes A, B are native of domain I and node C is native of domain II. Node B is a link node of domain I since it is connected with "ghost" node C. (b) Insertion of node D between B and C will make D the "new" link node and B the "old" link node. Thus (B, D, C) forms a Triplet of (`old`, `new`, `ghost`). Domain II will received this Triplet from domain I and update the link C-B with link C-D.

6 Remesh

Function `Remesh` ensures even discretization of dislocation lines by removing and inserting 2-nodes (nodes that have two neighbors). Node removal will be handled first. Nodes that have both neighbors local and are closer to either one of the neighbors than a minimum distance (`minSeg`) will be deleted (by `RemoveNode`), unless it is otherwise marked for exemption. To avoid over-deletion, the two neighboring nodes of each deleted node are marked. If the two dislocation segments of a 2-node glides on two different planes, the node is also marked for exemption.

If two nodes are separated by more than `maxSeg`, a new node will be inserted between them (by `InsertNewNode`). Node insertion is allowed not only between two native nodes, but also between a native node and a "ghost" node (a node that belongs to a neighboring domain). The latter case is much more complicated and is handled by transmitting "Triplets" between neighboring domains, as discussed below.

First we introduce the notion of "link" node. It is defined as a native node that has one or more neighbors being "ghost" nodes, such as node B of domain I in Fig. 12(a). When a new node D is inserted between B and C as shown in Fig. 12(b) by domain I, domain II has to be notified of this event, in order to replace the link C-B with link C-D. For this purpose, the domain that inserts a new node between a link node and a "ghost" node (in this case domain I) needs to report a Triplet of nodal indices (`old`, `new`, `ghost`) to the neighboring domain (domain II).

Before handling remeshing, `CommSendLinkNodes` and `CommUnpackLinkNodes` update the positions of neighboring "ghost" nodes of each link node, as they may be changed after nodal movement. `SetupTriplets` then allocates buffers for Triplet information exchange. After `Remesh`, Triplet information is sent between neighboring domains by `CommSendTriplets`. Link updates across domains such as depicted in Fig. 12(b) will then be performed by `FixTriplets`.

7 Load Balancing

7.1 Function Migrate

After motion some nodes leave the spatial limits of their original domain and need to be incorporated into a new domain, i.e. stored in a new processor; they are called "Migrators". Function `Migrate` handles nodal migration, which is listed below.

```
void Migrate (Home_t *home)
{
/* Build a list of link nodes: i.e. local nodes with non-local neighbors */
   FindLinkNodes (home) ;

/* Find all local out-migrators. Also add any new link nodes caused by
```

```
 * migration to link nodes list
 */
   MigFindMigrators (home) ;

/* Trade migrators with neighboring domains */
   CommSendMigrators (home) ;
   MigUnpackMigrators (home) ;

/* Send the domain neighbors the old/new tag pairs for migrators incoming
 * to this domain.
 */
   CommSendOldNewTags (home) ;

/* from the old/new tag pairs supplied by neighbors, plus the set generated
 * by this domain, reorganize all the old/new tag info for quick access
 */
   MigBuildOldNewMaps (home) ;

/* Use the old/new maps to update all linkages to and from migrating nodes */
   MigUpdateLinks (home) ;

   return ;
}
```

Function `MigFindMigrators` finds all the nodes that will migrate to neighboring domains by checking their current position against domain limits. An array `*outMigFlag` is initialized to be all zeros; for each migrator index the corresponding entry of `*outMigFlag` is set to one. The migrators indices from current domain (regardless of destination domain) are stored in array `*allMigs`. They are also grouped according to the destination domain and kept in the `*outMigs` array of "RemoteDomain" structures.

`CommSendMigrators` and `MigUnpackMigrators` then send these migrators between domains. Migrator nodes are freed from their original domain and inserted into the `**nodeKeys` array of the new domain. These new nodes as well as their neighbors are still having their old Tags, i.e. reflecting their memory location in the original domain. Their new indices are listed in the `*newNodes` array of the new domain.

To update nodal connections due to Tag change, the new domain uses `CommSendOldNewTags` to broadcast the old and new Tags of each new incoming nodes to all neighbor domains. This is necessary because, as shown in Fig. 13, two nodes that belonged to the same domain could migrate to two different domains, so that all domains in the neighborhood need to be notified of this change. "Triplets" of old domain ID, old index and new index of each incoming nodes are collected in `*oldNewBuf` of "Home" and is sent to `*intBuf` of the "RemoteDomains" structures of neighboring domains.

Now each domain has accumulated a set of integer buffers (`*oldNewBuf` of its own and `*intBuf` of each of its "RemoteDomains") recording the Tag change of migrator nodes; the buffers are indexed according to the new domain of migrator nodes, as shown in Fig. 14. To make this information more accessible, function `MigBuildOldNewMaps` builds up a set of maps (`*oldNewMap`) that group the migrator nodes according to their old domain indices, i.e. it performs a "transpose" operation on the integer buffers. As shown in Fig. 15, the "Home" structure as well as all of its "RemoteDomains" has an `oldNewMap`, which groups the migrator nodes according to their old domain ID. Each map is an array of new tags, indexed by the old indices of migrator nodes.

Once the "Maps" are completed, `MigUpdateLinks` then update the old Tag of each migrator node to its new Tag, and change the connections (neighboring nodes) of each migrator and link node to their new Tags.

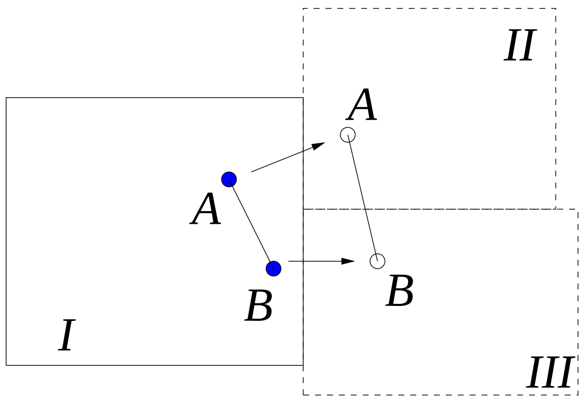


Figure 13: Two nodes A and B that belonged to domain I migrates to domain II and III respectively. All domains that are neighboring domain II and III will be notified of this change.

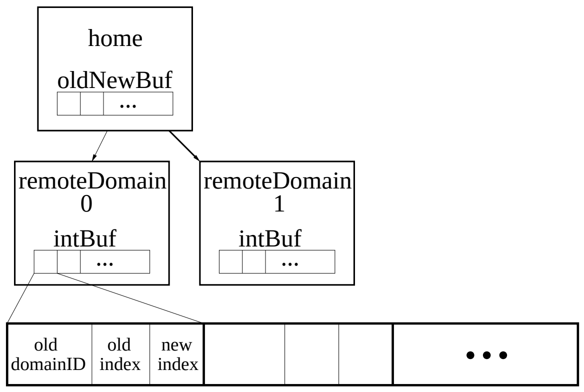


Figure 14: Integer buffers `*oldNewBuf` of "Home" and `*intBuf` of "RemoteDomains" group migrator nodes according to their new domain ID's. Each buffer entry is a triplet of `(old domainID, old index, new index)`.

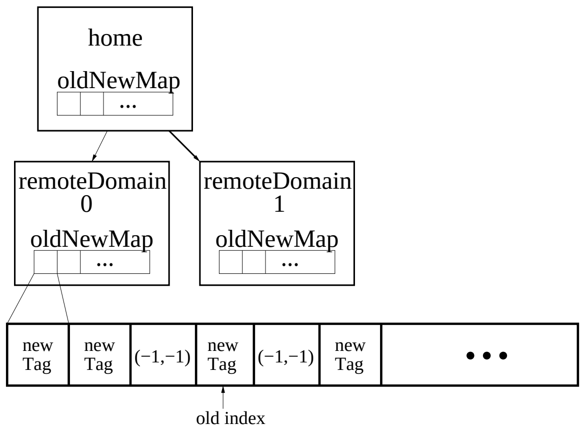


Figure 15: "Maps" are build based on integer buffers that group migrator nodes according to their old domain ID's. Each map is an array of new nodal "Tags" indexed by old nodal indices.

8 Loading Condition

9 Visualization

For debugging purposes, we installed an X-window display package to DD3d so that we can view dislocations during the simulation. `DisplayC.h`, `DisplayC.c` serve as C wrappers for the C++ class `YWindow` defined in `display.h`, `display.cpp`. In `DisplayC.h`, the following functions and control variables are exported to C callers. In a parallel code such as DD3d, the caller has to ensure that only a single processor (e.g. processor 0) calls `DisplayC` functions and manipulates window control variables.

```
/* DisplayC.h */
...

extern "C" void ReadWindowSpec(char *fname);

extern "C" void WinLock();
extern "C" void WinUnlock();
extern "C" void WinClear();
extern "C" int  WinAlive();
extern "C" void WinRefresh();
extern "C" void WinDrawPoint(double x,double y,double z,double r,
                             unsigned long color,unsigned long attr);
extern "C" void WinDrawLine(double x0,double y0,double z0,
                            double x1,double y1,double z1,
                            unsigned long color,
                            double r,unsigned long attr);
extern "C" void Sleep();
extern "C" unsigned long AllocShortRGBColor(unsigned r,
                                            unsigned g, unsigned b);
...

/* Window control variables */
extern int enable_window;
extern char win_name[100];
extern double point_radius, line_width;
extern int win_width, win_height;
extern int sleepseconds;
extern unsigned colors[MAXCOLOR];
extern char color_name[MAXCOLOR][COLORNAMELEN];
extern char bgcolor_name[COLORNAMELEN];
extern int color_scheme; /* 0: color by domain,
                            1: color by Burgers vector */
```

`ReadWindowSpec(char *fname)` reads window control variables from file specified by `*fname`, e.g. “`win.script`”. A sample “`win.script`” file is given below.

```
# win_script

enable_window = 1

win_width = 500  win_height = 500

color00 = red
color01 = green
color02 = magenta
color03 = cyan
#...
```

```
color_scheme = 1 #1 for domain, 2 for Burgers vector

point_radius = 0.005 line_width = 0.01

sleepseconds = 1000
```

The order of each data entry does not matter, as long as it complies with the syntax `varname = value`. A space has to exist before and after the `=` sign. The `#` sign comments out everything thereafter till the end of line. The binding between a string (e.g. ``‘‘win_width’’``) and the variable (e.g. `win_width`) is made possible by calling function `bindvar` in `DisplayC.c`, for example,

```
/* DisplayC.c */
...

void MyParser::initparser()
{
    ...

    bindvar("win_width",&win_width,INT);
    bindvar("win_height",&win_height,INT);

    ...
}
```

Functions `WinDrawPoint` and `WinDrawLine` allow user to draw points and lines into 3-dimensional space directly. The cubic framework in the viewing window has dimension `[-1,1]` in x, y, z directions. So coordination data need to be normalized to fit in the viewing window. Function `Plot` gives an example of how to use `DisplayC` routines.

```
/* Plot.c */
...

void Plot (Home_t *home)
{
    ...

    if(!WinAlive()) return;
    WinLock();
    WinClear();

    /* Find simulation box size Lx, Ly, Lz */

    ...

    /* Draw Nodes */
    for(i=0;i<home->newNodeKeyPtr;i++)
    {
        node=home->nodeKeys[i];
        if(node==0) continue;
        x=node->x; y=node->y; z=node->z;

        /* Normalize coordinates */
        x=(x-xmin)/Lx*2-1;
        y=(y-ymin)/Ly*2-1;
        z=(z-zmin)/Lz*2-1;
```

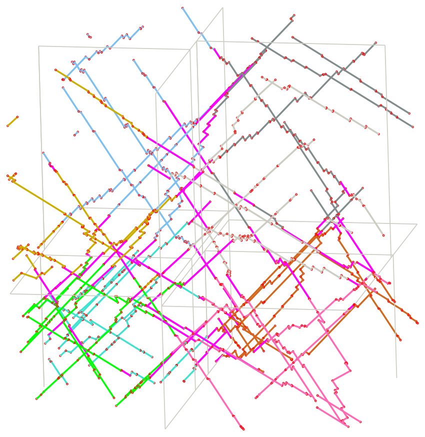


Figure 16: Screen shot from X-window plotting of DD3d. The entire simulation space is divided into 8 domains.

```
        WinDrawPoint(x,y,z,point_radius,colors[0],1);
    }

    WinUnlock();
    WinRefresh();
}
```

In DD3d, only processor 0 opens an X-window and plots the dislocation configuration. In order to show all the nodes, some of which are stored at other processors/domains, processor 0 has to receive all nodal information before plotting. This is done by calling `CommSendMirrorNodes` and `MirroUnpackNodes` at the beginning of function `Plot`. They store positional information of all nodes from other domains under the `MirrorDomain` structure of "Home". Fig. 16 is a screen shot of a DD3d configuration running with 8 processors (8 domains). The dislocation segments are colored according to the domains they belong to.

10 How to Use

10.1 Compile and Run

DD3d has been tested on Linux i386, Linux alpha, and ASCI Blue machines. To compile, type

```
make
```

It needs `MPI`, `X11` libraries and header files. To run, type

```
make run
```

For machines that implements MPI by MPICH, `make run` runs `dd3d` by invoking `mpirun -np ‘./getprocnum‘ dd3d`, where `getprocnum` computes the total number of processors by multiplying `numXdoms`, `numYdoms`, and `numZdoms` specified in the input file (by default `control.script`).

10.2 Input Files

The major input file for DD3d is a script file, by default `control.script`, but a different file can be supplied in the command line. For debugging purposes, DD3d also reads in a `win.script` file, which specifies how the dislocation configurations should be plotted in an X-window. These two script files have similar formats. They have actually a free-format, in that the data entries do not have to follow a fixed sequence. Values are assigned to variables through the `varname =` *value* syntax. The `control.script` file recognizes control variables, such as `numXcells = 3`, `totalsteps = 100` etc. The `win.script` file recognizes other variables, such as `win_width = 500`. It will be further discussed in Section 6.

The pound sign `#` marks for comments in the script files. Anything from `#` to the end of line is neglected by the parser. Preprocessor macros are also supported. For example, if the file contains `#define syma symb`, the preprocessor will replace every `syma` in the script by `symb` before handing it to DD3d. A line `#include ‘‘con1.cn’’` will cause the preprocessor to paste the content of file `con1.cn` to replace itself.

The nodal configurations, including positions and connectivities, are specified by `config = [` *nodal data* `]`, where *nodal data* is a chunk of string with a fixed format, which deserves some more explanation. A sample `control.script` file is given below.

```
### DD3d simulation control file (use -*-shell-script-*- format)
repartition = 1
printOldIDs = 0

### DD3d configuration (use -*-shell-script-*- format)
config = [
### (1) Box X, Y, Z (in b )
   -17500.0     -17500.0  -17500.0
    17500.0      17500.0   17500.0
### (2) Burgers vector array (number)
8
###(b_id, bx, by, bz)
 1    0.5773503    0.5773503    0.5773503
 2   -0.5773503   -0.5773503   -0.5773503
 3    0.5773503    0.5773503   -0.5773503
 4   -0.5773503   -0.5773503    0.5773503
 5    0.5773503   -0.5773503    0.5773503
 6   -0.5773503    0.5773503   -0.5773503
 7    0.5773503   -0.5773503   -0.5773503
 8   -0.5773503    0.5773503    0.5773503
### (3) Number of nodes
2
### (Primary lines: node_id, old_id, x, y, z, numNbrs, constraint, domain, index)
### (Secondary lines:   nbr[i], burgID[i])
```

```
     1      1    -2000.0000        10.0000     -8000.0000   1   7    0     0
                                 2      6
     2      1    -2000.0000        10.0000      8000.0000   1   7    0     1
                                 1      5
###domain boundaries (nXdoms, nYdoms, nZdoms)
   1    1    1
###domain boundaries (X,   Y,    Z)
   -17500.0
           -17500.0
                   -17500.0
                    17500.0
            17500.0
    17500.0
]

numXcells = 3 numYcells = 3 numZcells = 3
numXdoms  = 1 numYdoms  = 1 numZdoms  = 1
maxstep = 200
checkcollision = 1  ###0 no check, 1 check, 2 check and pause, 3 pause every step
                    ###4 check and highlight
eRate = 1.0  indxErate = 1 deltaTT = 1.0e-6
rmax = 200  rann = 10
appliedStress = [  1.e7 0 0  0 0 0 ]
```

The first line is a comment. The second line tells DD3d to repartition the nodes into different processors (domains). If on the other hand `repartition = 0` is given, then the previous partition given in the `config = [ ... ]` entry will be reused. The data within the bracket of `config = [ ... ]` follows a fixed format, although anything between `#` and the end of line is also neglected as a comment. The first six numbers specify the dimension of the simulation box, i.e $x_{\min}$, $y_{\min}$, $z_{\min}$, $x_{\max}$, $y_{\max}$, $z_{\max}$. Then follows the Burgers vector list, beginning with its length. After that, the total number of nodes is given, followed by the information of every node. Every node will take at least two lines. The first line gives its index (only meaningful within this file to specify connections), old index (for debugging purposes), positions (x, y, z), number of neighbors `numNbrs`, constrain type, domain ID and index within the domain. If `repartition = 0` was specified, the node will be given to the same domain and assigned to the same Tag as specified by the last two integers. Every node will then have `numNbrs` number of lines, giving the index of its neighbors and the Burgers vector ID for the segment. The following lines suggest how the simulation box should be decomposed into domains. They will be ignored if `repartition = 1` was given.

Outside the `config = [ ... ]` block, the script is free format again. In this script, the simulation box is divided into $3 \times 3 \times 3$ cells, but uses a single domain. The maximum simulation step is given by `maxstep = 200` . Variable `checkcollision` is used for debugging purposes. When it is 0, then no collision is checked and dislocation segments will simply pass through each other without experiencing short range reaction. If it is 1, collision detection and handling is applied. If it is 2, the simulation will pause whenever a collision is detected, and one will need to click 'p' in the X-window to resume the simulation. When it is 3, the simulation pauses at every step. For `checkcollision = 4`, the simulation does not pause, but will slow down whenever a collision is detected and nodes involving the collision will be highlighted.

There are other variables that can be specified in the `control.script` file. If they do not appear, then their default values will be used.

10.3 Output Files

Need to Write: Restart files, having the same format as `control.script`, and can be supplied to run DD3d again.

Dump files
`dump0_Before_Migrate`
`dump0_After_Migrate`
`dump1_Before_Migrate`
`dump1_After_Migrate`

A List of Control Variables

Need to Write: List of Functions

Control variables in structure `param` for `control.script`.

```
typedef enum {Periodic=0, Free=1, Reflecting=2} BoundType_t ;
typedef struct _param Param_t ; struct _param {
   int nXcells, nYcells, nZcells ; /* numXcells, numYcells, numZcells */
   int nXdoms, nYdoms, nZdoms ;    /* numXdoms,  numYdoms,  numZdoms  */

   BoundType_t xBoundType ; /* Periodic, Free, or Reflecting */
   BoundType_t yBoundType ;
   BoundType_t zBoundType ;
   ...
   real8 fixed ;
   real8 eRate ;            /* strain rate */
   real8 edotdir[3];        /* uniaxial loading axis */
   ...
   real8 Lx, Ly, Lz ;       /* length of entire simulation box */
   real8 minSeg ;  /* min allowable segment length, before removing a node */
   real8 maxSeg;   /* max allowable segment length, before adding a node   */
   ...
   real8 timeStart ;        /* Initial Time */
   real8 timeNow;           /* current time */
   int   allowSubcycling ;  /* if 0, recompute remote force every collision */
                            /* if 1, don't recompute every cycle necessarily */
   real8 nextRemForceTime ; /* time to recalc remote forces on all segments */
   ...
   real8 deltaTT;  /* proposed time step */
   real8 realdt;   /* real time step = min (deltaTT, tc_min); */
   ...
   int loadType ;  /* 0 Creep
                    * 1 Constant strain rate
                    * 2 Displacement-controlled
                    * 3 Load-controlled, load vs. time curve */
   real8 velcutoff; /* upper bound of all segment velocities (sound barrier) */
   real8 rmax ;    /* maximum migration distance per timestep for any node */
   real8 rann ;    /* closest distance before dislocations are
                    * considered in contact */
   real8 rc ;      /* core radius in elastic interaction calculation */
   real8 Ecore;    /* core energy (w.r.t. the choice of rc) in unit of Pa */
   int subdiv;     /* subdivision on segments (for Neighbor interaction) */

   int   cycleStart ;
   int   maxstep;

   real8 disloDensity; /* dislocation density */
   real8 shearModulus, pois, YoungsModulus; /* elastic constants */
   real8 burgMag;      /* length unit */
   real8 MobScrew;
   real8 MobEdge;
   real8 CheckScrew;
   int checkcollision;
   int elasticinteraction;
   int repartition ;
   int printOldIDs ;

   real8 appliedStress[6];
   real8 delpStrain[6],delSig[6],totpStn[6];
   real8 delpSpin[6],totpSpn[6];

   real8 dslipstn[12][3][3];
   real8 dslipspn[12][3][3];
   real8 slipstn[12][3][3];
   real8 slipspn[12][3][3];
   real8 dslipdens[12],olddens[12];
   real8 slipdens[12];

   int extendconfigspec[3] ;
   char micro3dcnfile[100] ;

    /* controls for various output files */

    int savecn,  savecnfreq,  savecncounter;
    int gnuplot, gnuplotfreq, gnuplotcounter;
    int tecplot, tecplotfreq;
    int psfile,  psfilefreq,  psfilecounter;
    int velfile, velfilefreq, velfilecounter;
    int armfile, armfilefreq, armfilecounter;
    int savedensityspec[3];
    int writepovrayspec[3];

    int saveprop, savepropfreq, savepropdetail;

    /* file directory name */
    char dirname[300];

    double sessileburgspec[30];
    double sessilelinespec[30];
    double calplanestressspec[10];
    double createconfigspec[10];
    int imgstrgrid[6];

    /* binary restart file name, and flag to specify binary or text restart
     * config data */

    char binfile[100] ;
    int binrestart ;

    char Rijmfile[100], RijmPBCfile[100];
    int readRijmfile ;
} ;
```

A sample `win.script` file.

```
enable_window = 0 #or 1
win_width = 500  win_height = 500 win_name = DD3d
```

```
rotateangles = [ 0 0 0 1.33 ] #Hor, Ver, Spin, Scale

color_scheme = 2 #1 for Domain, 2 for Burgers vector
bgcolor = grey30
color00 = red  #normal node color
color01 = magenta #cross link color
color02 = turquoise
color03 = green
color04 = SkyBlue2
color05 = gold3
color06 = chocolate
color07 = yellow
color08 = azure4
color09 = ivory3
color10 = ivory3
reversergb = 1

point_radius = 2 line_width = 2
sleepseconds = 3
```

B Nodal Driving Force Calculation

C Predicting Segment Pair Collision

Need to Write: Use bounding box to pre-screen segment pairs that are well separated. The need for a second cell structure.

In DD3d, we formulate the segment pair collision detection as a minimization problem. Given the initial nodal positions $\vec{r}_1, \vec{r}_2, \vec{r}_3, \vec{r}_4$ and velocities $\vec{v}_1, \vec{v}_2, \vec{v}_3, \vec{v}_4$, define

$$\vec{A}(\alpha,\beta,t) = [(\vec{r}_1+\vec{v}_1 t)(1-\alpha)+(\vec{r}_2+\vec{v}_2 t)\alpha] - [(\vec{r}_3+\vec{v}_3 t)(1-\beta)+(\vec{r}_4+\vec{v}_4 t)\beta] \quad (35)$$

$$F(\alpha,\beta,t) = \vec{A}(\alpha,\beta,t)\cdot\vec{A}(\alpha,\beta,t) \quad (36)$$

A collision is said to occur if

$$\min_{\substack{0\le\alpha,\beta\le 1\\ 0\le t\le\Delta t}} F(\alpha,\beta,t) \le \rho^2 \quad (37)$$

is satisfied. When this is the case, we need to determine the "projected" time t_p at which inequality (37) is first satisfied. ρ is introduced to circumvent the round-off error and to improve the robustness of the algorithm. At the same time, one can also consider ρ as the physical core radius of the dislocation; if two dislocation segments are closer than ρ they are technically in contact.

To solve this minimization problem, let's first look at the derivatives of function F and $\vec{A}$, where $\vec{A}_\alpha \equiv \partial\vec{A}/\partial\alpha$, $\vec{A}_{\alpha\beta} \equiv \partial^2\vec{A}/\partial\alpha\partial\beta$, etc.

$$\vec{A}_\alpha = (\vec{r}_2+\vec{v}_2 t)-(\vec{r}_1+\vec{v}_1 t) \quad (38)$$

$$\vec{A}_\beta = -(\vec{r}_4+\vec{v}_4 t)+(\vec{r}_3+\vec{v}_3 t) \quad (39)$$

$$\vec{A}_t = [\vec{v}_1(1-\alpha)+\vec{v}_2\alpha]-[\vec{v}_3(1-\beta)+\vec{v}_4\beta] \quad (40)$$

$$\vec{A}_{\alpha\alpha} = 0 \quad (41)$$

$$\begin{aligned}\vec{A}_{\beta\beta} &= 0 && (42)\\ \vec{A}_{tt} &= 0 && (43)\\ \vec{A}_{\alpha\beta} &= 0 && (44)\\ \vec{A}_{\alpha t} &= \vec{v}_2 - \vec{v}_1 && (45)\\ \vec{A}_{\beta t} &= -\vec{v}_4 + \vec{v}_3\ , && (46)\end{aligned}$$

The derivatives of F can be expressed in terms of the derivatives of $\vec{A}$.

$$\begin{aligned}F_\alpha &= 2\vec{A}\cdot\vec{A}_\alpha && (47)\\ F_\beta &= 2\vec{A}\cdot\vec{A}_\beta && (48)\\ F_t &= 2\vec{A}\cdot\vec{A}_t && (49)\\ F_{\alpha\alpha} &= 2\vec{A}_\alpha\cdot\vec{A}_\alpha && (50)\\ F_{\beta\beta} &= 2\vec{A}_\beta\cdot\vec{A}_\beta && (51)\\ F_{tt} &= 2\vec{A}_t\cdot\vec{A}_t && (52)\\ F_{\alpha\beta} &= 2\vec{A}_\beta\cdot\vec{A}_\beta && (53)\\ F_{\alpha t} &= 2\vec{A}_t\cdot\vec{A}_\alpha + 2\vec{A}\cdot\vec{A}_{\alpha t} && (54)\\ F_{\beta t} &= 2\vec{A}_t\cdot\vec{A}_\beta + 2\vec{A}\cdot\vec{A}_{\beta t} && (55)\end{aligned}$$

The Jacobian

$$J(\alpha,\beta,t) = \det\begin{pmatrix} F_{\alpha\alpha} & F_{\alpha\beta} & F_{\alpha t}\\ F_{\alpha\beta} & F_{\beta\beta} & F_{\beta t}\\ F_{\alpha t} & F_{\beta t} & F_{tt}\end{pmatrix} \qquad (56)$$

can be either positive or negative, so that in general $F(\alpha,\beta,t)$ is neither strictly convex or concave. However, for any given t, the sub-Jacobian with respect to α and β

$$\det\begin{pmatrix} F_{\alpha\alpha} & F_{\alpha\beta}\\ F_{\alpha\beta} & F_{\beta\beta}\end{pmatrix} = 4\det\begin{pmatrix}\vec{A}_\alpha\cdot\vec{A}_\alpha & \vec{A}_\alpha\cdot\vec{A}_\beta\\ \vec{A}_\alpha\cdot\vec{A}_\beta & \vec{A}_\beta\cdot\vec{A}_\beta\end{pmatrix} = 4(\vec{A}_\alpha - \vec{A}_\beta)^2 \geq 0 \qquad (57)$$

This means that for any given t, $F(\alpha,\beta,t)$ is convex with respect to α, β. This motivates us to minimize F with respect to α, β first, and lastly for t.

For a given t, one can show that $F(\alpha,\beta,t)$ is a quadratic function of α,β, with terms such as α, $\alpha\beta$ etc. Minimizing F with respect to α and β is straight forward. In general, the minimum will be achieved at $\hat{\alpha}$ and $\hat{\beta}$ which satisfies,

$$\begin{aligned}F_\alpha &= (\vec{r}_2-\vec{r}_1)\cdot(\vec{r}_1-\vec{r}_3) + (\vec{r}_2-\vec{r}_1)^2\alpha - (\vec{r}_2-\vec{r}_1)\cdot(\vec{r}_4-\vec{r}_3)\beta &= 0 && (58)\\ F_\beta &= (\vec{r}_4-\vec{r}_3)\cdot(\vec{r}_1-\vec{r}_3) - (\vec{r}_4-\vec{r}_3)\cdot(\vec{r}_2-\vec{r}_1)\alpha + (\vec{r}_4-\vec{r}_3)^2\beta &= 0 && (59)\end{aligned}$$

Here we write $\vec{r}_1$ to represent $\vec{r}_1 + \vec{v}_1 t$, etc. When the solution of Eqs.(59-59) lies outside of the domain $0 \leq \alpha,\beta \leq 1$, the optimal solution $(\hat{\alpha},\hat{\beta})$ then has to lie on the domain boundary. When this is the case, we choose $(\hat{\alpha},\hat{\beta})$ from the following four solutions.

Need to Write: Degenerate case: $det(F_{\alpha\beta}) = 0$, when $\vec{r}_{12} \parallel \vec{r}_{34}$

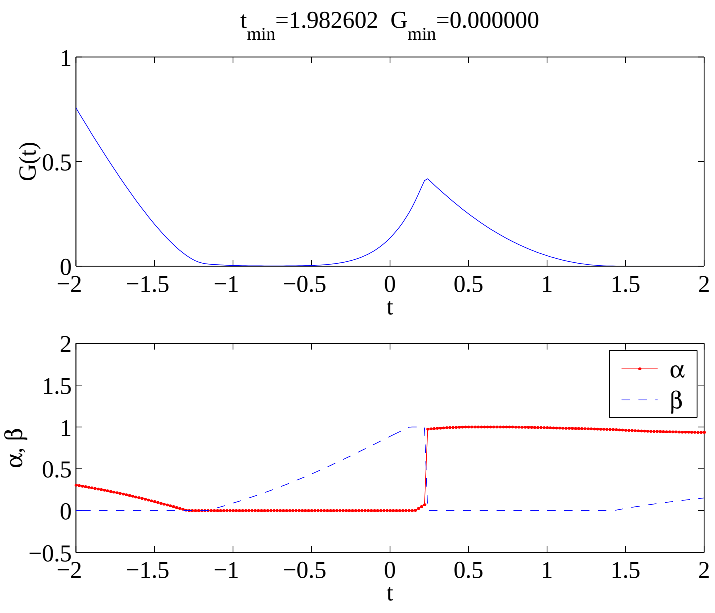


Figure 17: An example of (a) G(t) and the corresponding (b) $\hat{\alpha}(t)$, $\hat{\beta}(t)$.

1. $\alpha = 0$,
2. $\alpha = 1$,

(for 1 and 2)
$\beta' = (\vec{r}_4 - \vec{r}_3) \cdot [(\vec{r}_1 - \vec{r}_3) + (\vec{r}_2 - \vec{r}_1)\alpha]/(\vec{r}_4 - \vec{r}_3)^2$,
$\beta = \min(1, \max(0, \beta'))$

3. $\beta = 0$,
4. $\beta = 1$,

(for 3 and 4)
$\alpha' = (\vec{r}_2 - \vec{r}_1) \cdot [-(\vec{r}_1 - \vec{r}_3) + (\vec{r}_4 - \vec{r}_3)\beta]/(\vec{r}_2 - \vec{r}_1)^2$,
$\alpha = \min(1, \max(0, \alpha'))$

Therefore, for arbitrary t we can solve for the optimal $\hat{\alpha}(t)$ and $\hat{\beta}(t)$ which minimizes $F(\alpha, \beta, t)$ for the fixed t. Define

$$G(t) = F(\hat{\alpha}(t), \hat{\beta}(t), t) \tag{60}$$

An example of $G(t)$ and the corresponding $\hat{\alpha}(t)$, $\hat{\beta}(t)$ are plotted in Fig. 17. In this case, the two segments are moving on the same plane, so that $G(t)$ can be zero on a continuous domain of t. Also note that $G(t)$ is not monotonic and can have multiple local minimums.

Now Eq.(37) reduces to a one-dimensional problem, i.e. whether and at which t does $G(t)$ become smaller that ρ^2. We use the following "sawtooth" algorithm to solve this problem.

Define $g(t) = \sqrt{G(t)}$, the "sawtooth" algorithm will determine the first t in the domain $[0, \Delta t]$ for which $g(t) \leq \rho$. Since $g(t)$ is the closest distance between the two segments at time t, it has to satisfy

$$\frac{dg(t)}{dt} \leq v_{\max} \ , \tag{61}$$

where

$$v_{\max} = \max(|\vec{v}_1 - \vec{v}_3|, |\vec{v}_1 - \vec{v}_4|, |\vec{v}_2 - \vec{v}_3|, |\vec{v}_2 - \vec{v}_4|) \ , \tag{62}$$

because $v_{\max}$ is the upper bound to the relative velocity between any two points on the two segments.

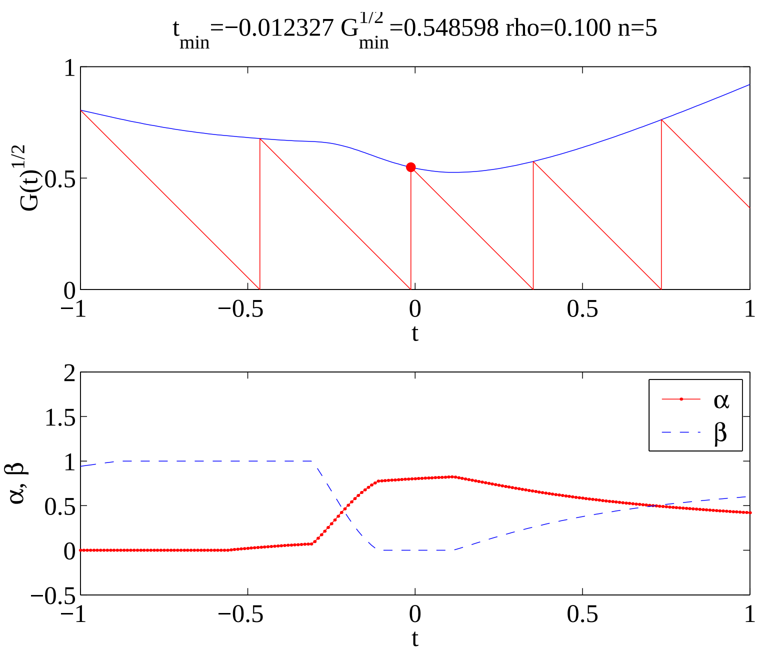


Figure 18: Example of sawtooth algorithm on [-1,1]. The algorithm concludes that no collision is possible after 5 evaluations of $G(t)$.

The "sawtooth" algorithm utilizes this information (the derivative of $g(t)$ is bounded by a known value) to scan a time domain $[t_0, t_1]$ and determine whether $g(t) \leq \rho$ is possible. The algorithm goes as the following.

1. Set $t = t_0$
2. If $t > t_1$, no collision possible, exit.
3. If $g(t) < \rho$, collision detected, exit.
4. Else $t = t + g(t)/v_{\max}$, go to 2.

The purposes of steps 1, 2, 3 of the above algorithm are obvious. In step 4, we increase our time by $g(t)/v_{\max}$ because if both $g(t_1) > \rho$ and $g(t_2) > \rho$, while $t_2 = t_1 + g(t_1)/v_{\max}$, then one can prove that for all $t \in [t_1, t_2]$, $g(t) > q/2$. In this case, one can claim (within some approximation) that Eq.(37) has no solution in domain $[t_1, t_2]$ and one can then examine the next domain $[t_2, t_2 + g(t_2)/v_{\max}]$.

When the two segments well separated from each other during $[t_0, t_1]$, the "sawtooth" algorithm quickly scans through the domain with only a few evaluations of $g(t)$, as shown in Fig. 18. When collisions exist, the algorithm will quickly converge to the first occurrence of the collision, as shown in Fig. 19.

Need to Write: Using the same function to detect type (2) and (3) events.

D Cost Analysis

`TimeStart()` and `TimeStop()`

An example timing report `time_0`.

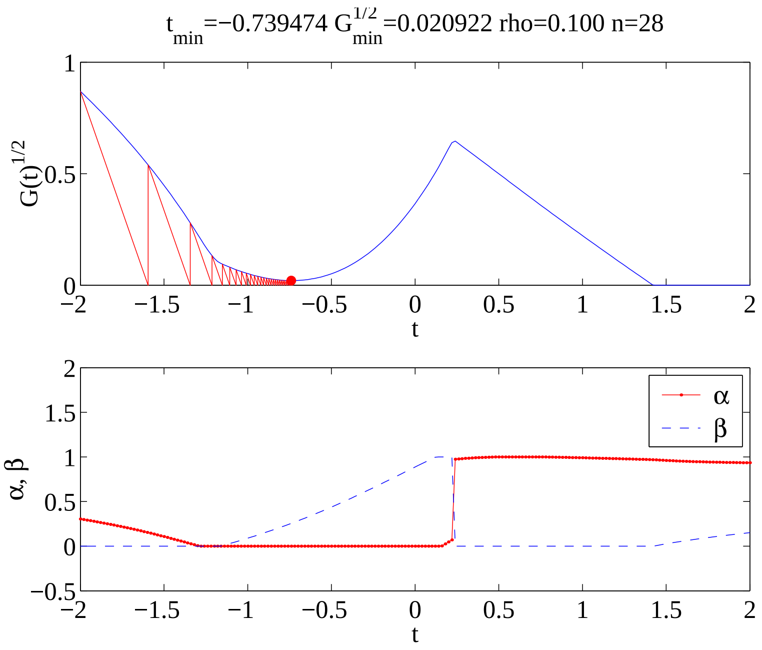


Figure 19: Example of sawtooth algorithm on [-1,1]. The algorithm converges to the first collision after 28 evaluations of $G(t)$.

```
domain : 0     total cycles : 200

total time                 13.421528 initialization time
0.088086 sort native nodes           0.003847 comm send ghosts
0.002677 calculate velocity          0.156644 comm send velocity
0.000579 advance and reconnect      12.136459 comm send reconnects
0.000308 fix reconnects              0.000327 X window plot
0.174117 comm send link nodes        0.003186 comm send triplets
0.000898 migration                   0.016039
```

References

References

[1] Wei Cai, *Atomistic and Mesoscale Modeling of Dislocation Mobility*, Ph.D. Thesis, M.I.T., May (2001).